MASTER EN SCIENCES INFORMATIQUES

Contribution au développement du langage de description d'architecture MADL à travers une étude de cas
définition d'un mapping technologique vers CORBA IDL

Karlik, Joachim

Award date:
2007

Contribution au développement du Langage de Description
d'Architecture MADL à travers une étude de cas :

Définition d'un Mapping technologique vers CORBA IDL

Joachim KARLIK

Sous la direction de Vincent ENGLEBERT

Facultés Universitaires Notre Dame de la Paix, Institut d'Informatique, Namur

Septembre 2007

Résumé

Ce mémoire propose une contribution à la définition du langage de description d'architecture MADL. Premièrement, nous définissons les concepts de base du langage en leur associant une sémantique intuitive. Nous proposons ensuite une validation des choix conceptuels opérés lors de cette définition par le développement d'une solution de Mapping technologique vers CORBA IDL. Nous établissons pour cela des pistes de solutions permettent le déploiement d'une architecture décrite en MADL sur un système distribué. Les règles de Mapping sont ensuite validées par prototypage. Finalement, nous proposons, sur base d'une étude bibliographique, des mécanismes de reconfiguration dynamique de la structure d'une architecture pour le langage MADL et critiquons notre approche dans le cadre d'un exemple bien précis.

Abstract

This study aims at providing a contribution for the definition of an Architecture Description Language, named MADL. Firstly, we introduce the basics of the language and associate an intuitive semantic to the main concepts. We propose then to validate the conceptual decisions made during that definition by developping a technologic Mapping solution towards CORBA IDL. In order to reach that goal, we establish some pists of solutions to allow the deployment of an architecture (described with MADL) on a distributed environment. The Mapping rules are validated by a prototype. Finally, we propose, with the help of some others studies, some mecanisms of structural dynamic reconfiguration for the language MADL and bring a critique of our approach within the framework of an example of a specific dynamic reconfiguration.

Remerciements

Mes remerciements vont à Vincent Englebert, pour sa disponibilité, le temps passé à m'expliquer, ses conseils et nos discussions.

Table des matières

Table des figures

Introduction

La conception de logiciels fait face à de nombreuses difficultés telles que la gestion de la réusabilité, de l'interopérabilité, ou encore de la flexibilité. La programmation orientée objet apporte certaines solutions à ces problèmes en permettant une certaine abstraction des tâches opérationnelles servant à décrire un système. Cependant, cette méthode encombre le développement d'applications de détails encore trop techniques propres à l'environnement comme le multi-threading ou la gestion de la sécurité par exemple. Afin de rendre la tâche du développement logiciel moins complexe, il est nécessaire de disposer d'un niveau d'abstraction encore plus élevé en se basant sur des modèles plus proches de la conception mentale du développeur. Une solution possible est de définir ce que nous appelons une architecture du système.

Le domaine relatif à l'étude des architectures des logiciels est assez jeune. Même si nous utilisons depuis longtemps de nombreux concepts de base lui appartenant, ce domaine n'existe réellement que depuis dix ans. En effet, l'ouvrage de Shaw et Garlan, *Software Architecture : Perspectives on a Emerging Discipline* [1], constitue le point de départ de l'étude des architectures des logiciels et son acceptation comme nouvelle discipline interne à celle qui étudie le développement de logiciels. Shaw et Garlan insistent sur le fait que la conception d'une architecture devient incontournable lorsque la taille et la complexité du système augmente. De nombreux ouvrages ont ensuite apporté leur définition d'architecture. Il s'avère que la plupart de ces définitions n'entrent pas en conflit et s'accordent sur le fait qu'une architecture représente la structure globale d'un système constitué de composants et de relations entre eux. Shaw et Garlan définissent une architecture comme étant "un niveau de design qui implique la description d'éléments (à partir desquels des systèmes sont construits), des interactions entre ces éléments, des modèles qui guident leur composition ainsi que la description des contraintes sur ces modèles". Dans *Software Architecture in Practice*, une architecture est définie comme étant "une structure du système qui comprend les composants logiciels, les propriétés externes visibles de ces composants ainsi que les relations qui existent entre eux" [2].

Une architecture constitue donc un plan du logiciel qu'il est possible de construire à l'aide de certains outils formels capables de générer du code automatiquement. Les langages de description d'architecture (ou ADLs pour *Architecture Description Languages*) font partie de ces outils. Ils fournissent aux développeurs un support d'aide à la modélisation leur permettant de spécifier les composants et les interactions qui peuvent exister entre eux. Cette spécification est faite de manière abstraite et ne comporte aucun détail d'implémentation.

De nombreux langages de description d'architecture ont été développés et adressent des problèmes spécifiques à la description d'architecture. Cependant, aucun d'entre eux ne permettent la description d'une architecture à différents niveaux d'abstraction. MADL est un langage de description d'architecture dont l'objectif est de combler cette lacune. Toujours en cours de définition, ce langage ne possède encore aucune sémantique. Les concepts de base du langage ont été fixés et une syntaxe concrète définie.

L'enjeu de ce mémoire est de contribuer à la définition du langage MADL en axant notre méthode de travail autour de trois objectifs principaux :

* **Validation des choix conceptuels du langage** : A travers un cas d'étude, nous allons valider les choix conceptuels qui interviennent dans la définition du langage MADL. Il s'agit de valider ces choix par un essai de mapping technologique. En effet, nous allons essayer d'établir des règles de transformations de MADL vers CORBA IDL. Ces règles ont pour but de mettre en oeuvre des mécanismes de déploiement d'une architecture décrite en MADL sur un système réparti. L'élaboration de telles règles nécessitent qu'une certaine sémantique soit associée aux concepts de base du langage.

* **Validation des règles de mapping par prototypage** : Le second objectif consiste à valider l'ensemble des règles de mapping élaborées en les implémentant au sein d'un prototype. Ce prototype doit être capable de déployer une application répartie à partir de la description de son architecture.

* **Reconfiguration dynamique** : Finalement, nous utiliserons les résultats obtenus afin de concevoir des pistes de solutions permettant de doter le langage MADL de mécanismes de reconfiguration dynamique d'une architecture. Ce troisième objectif représente la partie originale de ce mémoire.

Ce travail s'organise en cinq chapitres :

* **Le chapitre 1** : présente un état de l'art sur les langages de description d'architecture. A la fin de ce chapitre, le lecteur aura une vision d'ensemble des objectifs et concepts de base d'un ADL.

* **Le chapitre 2** : présente en détail les concepts de base du langage MADL en associant une sémantique intuitive aux principales structures du langage.

* **Le chapitre 3** : propose des pistes de solutions dans le cadre d'un mapping technologique vers le middleware CORBA IDL.

* **Le chapitre 4** : démontre par prototypage la faisabilité des règles élaborées dans le chapitre 3. Les différentes phases de développement du prototype sont détaillées dans ce chapitre.

* **Le chapitre 5** : propose des mécanismes de reconfiguration dynamique pour le langage MADL. Ces solutions se basent sur une recherche bibliographique. A la fin de ce chapitre, les règles de mapping sont critiquées dans le cadre de la mise en oeuvre de certaines reconfigurations dynamiques.

Chapitre 1

Etat de l'art sur les Langages de Description d'Architecture (ADLs)

1.1 Les concepts de base des ADLs

Dans cette section, nous décrivons les trois concepts de base présents dans la plupart des langages de description d'architecture : le *composant*, le *connecteur* et la *configuration d'architecture*.

Dans le document intitulé *A Classification and Comparison Framework for Software Architecture Description Languages*[3], N.Medvidovic et R. Taylor définissent, pour chacun de ces trois éléments, les principales caractéristiques communes aux ADLs afin de permettre leur comparaison. Cette taxonomie permet aussi d'évaluer un ADL et de considérer les caractéristiques qu'un langage doit posséder pour être considéré comme un langage de description d'architecture.

Dans la suite de cette section, nous utilisons toutes ces caractéristiques pour définir chaque élément de base. Même si il existe de nombreux documents concernant la définition et la classification des ADLs, le travail de N. Medvidovic et R. Taylor représente une source assez complète et pertinente en matière d'état de l'art des langages de description d'architecture. En effet, la plupart des langages de description d'architecture existants ont été étudiés afin d'établir ces critères de comparaison des ADLs.

1.1.1 Le composant

Un composant dans une architecture est une unité de calcul ou de stockage. Il peut être soit simple soit composé d'autres composants. Dans ce dernier cas, nous utilisons le terme de composant *composite*. Un composant se divise en deux parties distinctes : la partie *externe* qui spécifie les interfaces fournies et requises par le composant et la partie *interne* qui décrit son implantation en spécifiant le comportement interne du composant. Les caractéristiques de comparaison de N. Medvidovic et R. Taylor pour la modélisation des composants sont l'*interface*, le *type de composant*, la *sémantique*, les *contraintes*, l'*évolution* et les *propriétés non fonctionnelles*. Chacune de ces caractéristiques est discutée ici.

L'interface d'un composant constitue un point d'interaction avec le monde extérieur. En effet, l'interface définit soit les services rendus par un composant ou soit les services dont un composant a besoin pour fonctionner correctement. Un service peut correspondre à un message, une opération ou une variable.

Les types de composant sont des abstractions décrivant un ensemble de fonctionnalités pouvant être réutilisées. Un type de composant peut être instantié plusieurs fois. Toutes les instances partagent les propriétés encapsulées dans le type de composant.

La sémantique d'un composant peut être définie comme étant un modèle de son comportement. Outre la définition de son interface, ce modèle abstrait permet de décrire le comportement dynamique

d'un composant ainsi que les contraintes liées à l'architecture. Un tel modèle est nécessaire pour assurer correctement la transformation de l'architecture d'un niveau d'abstraction à un autre.

Les contraintes sont des propriétés d'un système qui doivent absolument être vérifiées pour que le système soit considéré comme acceptable. Placer des contraintes sur un composant permet de spécifier ses dépendances internes comme la gestion du nombre d'instances d'un certain type par exemple.

L'évolution d'un composant peut être définie comme une modification de ses propriétés dans le temps. Une propriété d'un composant peut être ses interfaces, sa sémantique, ses contraintes, ou encore son implémentation. Un langage de description d'architecture doit être capable d'assurer l'évolution d'un composant de manière systématique.

Les propriétés non fonctionnelles, comme la sécurité ou la qualité de service par exemple, doivent être spécifiées de manière distincte par rapport à l'aspect métier d'un composant. De telles propriétés permettent de se rendre compte dès la conception du comportement que peut avoir un composant à l'exécution et de valider l'architecture lors de son déploiement dans un environnement d'exécution.

1.1.2 Le connecteur

Les connecteurs sont des éléments de base de construction d'architectures utilisés par les ADLs afin de modéliser les interactions entre les composants et les règles qui gouvernent ces interactions. Ils contrôlent la communication et les activités de coordination entre plusieurs composants. Par exemple, un connecteur peut être utilisé pour décrire de simples interactions comme un appel de méthode ou pour décrire des communications plus complexes comme la diffusion d'évènements asynchrones vers plusieurs composants désirant recevoir ces évènements. Les connecteurs sont caractérisés par leur *interface*, leur *type*, leur *sémantique*, leurs *contraintes*, leur *évolution* et leurs *propriétés non fonctionnelles*.

L'interface d'un connecteur décrit l'ensemble des interactions qu'il possède avec le monde extérieur, à savoir d'autres composants et connecteurs qui lui sont attachés. Contrairement à un composant, le connecteur n'exécute aucun calcul spécifique à l'application mais expose plutôt les services des composants auxquels il est rattaché.

Les types de connecteur sont des abstractions décrivant un ensemble de mécanismes qui spécifient la manière dont les composants peuvent communiquer ensemble. Ces mécanismes forment ce que l'on appelle un protocole d'interaction. Afin de permettre leur réutilisation, les ADLs modélisent les connecteurs en tant que types. Par exemple, un connecteur de type 2PC définit les règles du protocole "*Two-phase commit*" permettant la gestion de la validation des transactions dans un système distribué.

La sémantique d'un connecteur peut aussi être définie comme étant un modèle qui spécifie son comportement. En d'autres mots, la sémantique d'un connecteur a pour but de spécifier son protocole d'interaction en vue de valider la transformation de l'architecture d'un niveau abstrait vers son implémentation.

Les contraintes d'un connecteur permettent de placer une limitation sur son utilisation. Un exemple de contrainte est de spécifier un nombre maximum limitant l'interconnexion de composants à travers un connecteur.

L'évolution d'un connecteur peut être définie comme une modification de son interface ou de son comportement. Les protocoles de communication qui régissent les règles d'interaction entre les composants d'un système ont tendance à changer ou à évoluer. C'est pourquoi un ADL doit être capable de modifier ou de raffiner un connecteur existant afin de garantir une concordance entre le système et un protocole d'interaction ayant subi un changement.

Les propriétés non fonctionnelles d'un connecteur sont des besoins additionnels qui ne sont pas spécifiés par sa sémantique. Elles permettent une implémentation correcte de ce connecteur

1.1.3 La configuration d'architecture

Une configuration d'architecture définit la structure d'une application à l'aide de composants et de connecteurs. Nous pouvons distinguer deux types de configurations : la configuration *structurelle* et la configuration *comportementale*. La configuration structurelle désigne un ensemble de composants interconnectés. Elle a pour rôle de vérifier si les interfaces des composants et des connecteurs correspondent. La configuration comportementale modélise le comportement général de l'application. Elle permet de vérifier si la sémantique combinée des composants et des connecteurs résulte bien en un comportement souhaité. Par exemple, elle peut définir la façon dont les connecteurs peuvent évoluer ou encore la façon dont les composants sont instantiés à l'initialisation de l'application. N. Medvidovic et R. Taylor identifient les caractéristiques suivantes pour une configuration d'architecture : le *formalisme*, la *composition*, le *raffinement et la traçabilité*, l'*hétérogénéité*, le *passage à l'échelle*, l'*évolution*, le *dynamisme*, les *contraintes* et les *propriétés non fonctionnelles*.

Le formalisme utilisé par un ADL pour modéliser la structure d'une architecture doit utiliser une syntaxe compréhensible et simple afin de faciliter la communication entre tous les partenaires d'un projet. Idéalement, la structure d'un système devrait pouvoir être comprise facilement sans devoir étudier la spécification des composants et des connecteurs.

La composition est un mécanisme permettant aux ADLs de définir la structure d'un système à différents niveaux d'abstraction. Ainsi un composant peut être composé d'autres composants. Un composant qui ne peut être décomposé est dit primitif. Ce mécanisme permet aux concepteurs d'une architecture de créer sa structure par raffinement descendant, c'est à dire du niveau le plus abstrait jusqu'au niveau le plus détaillé.

Le raffinement d'une architecture consiste à passer d'un niveau abstrait vers un niveau plus détaillé d'une architecture. Les ADLs doivent être capable de raffiner la structure d'un système en différents niveaux d'abstraction, de relier ces différents niveaux de raffinement et de garder trace des changements effectués à travers ces niveaux. Ce mécanisme de raffinement et de traçabilité est nécessaire pour rapprocher les modèles de haut niveau des langages de programmation.

L'hétérogénéité des composants et des connecteurs est une caractéristique importante que les ADLs doivent prendre en compte. En effet, ils doivent être capable de définir la structure d'un système sans tenir compte de la nature des composants et des connecteurs, du langage de programmation dans lesquels ils sont implémentés ou encore du système d'exploitation sur lequel ils sont déployés.

Le passage à l'échelle se traduit par une augmentation de la complexité et de la taille d'un système dans le temps. Les ADLs doivent fournir des mécanismes de modélisation qui permettent de définir des systèmes susceptibles de gagner en taille et en complexité.

L'évolution. Une architecture évolue pour refléter les changements apportés à un système. L' évolution d'un système (exemple : la maintenance) étant une activité importante dans le processus de développement d'une application, une configuration doit pouvoir faire évoluer l'architecture simplement en ajoutant ou retirant des composants ou des connecteurs.

Le dynamisme. Alors que l'évolution se traduit par des changements "off-line" de l'architecture, le dynamisme se traduit plutôt par des changements de l'architecture alors que le système est en cours d'exécution. Dans le cas de certains systèmes critiques, il serait inadmissible de stopper leur exécution pour appliquer ces changements. C'est pourquoi les ADLs doivent prévoir des mécanismes pour modéliser le dynamisme d'une application et appliquer ces changements sur le système en exécution.

Les contraintes qui décrivent les limites de la configuration dépendent en partie ou directement des contraintes liées aux composants et aux connecteurs. Ces contraintes, appelées aussi contraintes globales, concernent l'interaction entre les composants et les connecteurs.

Les propriétés non fonctionnelles qui ne concernent ni les composants ni les connecteurs sont exprimées au niveau de la configuration et sont liées à l'environnement d'exécution. Les ADLs doivent permettre la définition de telles contraintes.

1.2 Exemples de langages de description d'architecture

L'ensemble des langages de description d'architecture peut être divisé en deux grandes catégories. D'une part, nous avons les langages de description d'architecture. Ils s'intéressent essentiellement à définir la structure d'un système en assemblant des éléments entre eux. Cependant, ces langages ne peuvent spécifier l'aspect dynamique d'un système, ils sont dits *statiques*. D'autre part, nous avons les langages de configuration d'architecture qui s'intéressent plutôt à modéliser une configuration d'un système en termes d'instances de types de composants. En outre, certains sont capables de spécifier son comportement dynamique avant, pendant et après son exécution. Ces langages permettent la définition du cycle de vie d'un système et sont dits *dynamiques*. MADL étant un langage destiné à faire partie de cette deuxième catégorie, nous trouvons plus pertinant de vous présenter en détails deux ADLs importants qui en font aussi partie, à savoir *Darwin* et *Fractal*. Le lecteur intéressé pourra trouver une étude complète des différents ADLs existants dans le document intitulé *Etat de l'art sur les Langages de Description d'Architecture (ADLs)* [4].

1.2.1 Darwin

Darwin [4] est un langage de configuration permettant de décrire la structure et la configuration d'une architecture mais aussi de définir son comportement à l'exécution. L'originalité de ce langage se trouve dans l'utilisation de composants instantiables. En effet, un composant est une unité qui peut être instantiée à de multiples reprises lors de l'exécution de l'application. Ce langage se distingue donc des autres ADLs, comme Unicom par exemple, qui ne possèdent pas cette notion d'instances et sont donc incapables de prévoir le comportement d'une architecture modélisée à son exécution.

Darwin définit une configuration comme une hiérarchie d'instances de composants interconnectées entre elles. Chaque niveau de la hiérarchie est représenté par des instances de types de composants composites qui décrivent des interconnexions entre composants tandis que le dernier niveau correspond à des instances ne comportant pas de sous configurations. Typiquement, ces dernières, appelées instances primitives, représentent les "véritables" modules de l'application qui intègrent le code logiciel. Elles sont instantiées depuis des types de composants primitifs. L'implémentation des interfaces exposées est à charge du programmeur.

Dans cette section, nous allons décrire plus en détails les concepts importants et particularités de ce langage afin de comprendre comment construire la configuration d'un système. Nous illustrerons ces concepts grâce à un exemple avant de conclure sur les principaux avantages et inconvénients du langage Darwin.

La description du langage Darwin

En général, les ADLs peuvent attribuer différentes sémantiques à la notion de composant. Cependant Darwin n'en associe qu'une seule : la sémantique du *processus*. Un tel choix fut motivé par l'appropriation des processus à la programmation concurrente, distribuée ou encore parallèle. De cette manière, à chaque nouvelle instance d'un type de composant, un nouveau processus est créé. Ces processus ont la capacité d'être paramétrables à l'initialisation. Par exemple, Darwin permet à un composant composite de décrire sa configuration dynamiquement à l'initialisation en spécifiant le nombre d'instances des types de composants qu'il encapsule. Ce mécanisme permet ainsi de spécifier entièrement le schéma de création des instances à l'initialisation d'un système.

Chaque composant possède un type. Ce type représente une abstraction à partir de laquelle plusieurs instances peuvent être créées. La déclaration d'un type de composant s'effectue grâce au mot clé **component**. L'instruction suivante déclare un composant de type A :

```
component A;
```

A partir de ce type de composant, une ou plusieurs instances peuvent être créées grâce au mot clé **inst** de la manière suivante :

```
inst I1:A; I2:A; I3:A;
```

Le mécanisme qui consiste à déclarer des instances est appelé *instanciation de composant*. En outre, Darwin fournit les clauses *when* et *forall* qui permettent de définir des schémas d'instanciation plus raffinés. Nous verrons dans un exemple comment sont utilisées ces deux instructions.

Même si cette façon de déclarer des composants et des instances est un moyen très simple pour comprendre la manière dont on construit les différents blocs d'une application, elle est aussi beaucoup trop simpliste que pour pouvoir décrire les interconnexions existantes entre ces différentes parties. Un besoin important concernant ces interconnexions est de permettre la déclaration de composants de manière indépendante par rapport à la configuration. Ainsi, Il doit être possible de spécifier des composants sans que ce dernier ait besoin de savoir avec quels autres composants il devra interagir par la suite. C'est la raison pour laquelle un composant possède une ou plusieurs interfaces d'interaction avec le monde extérieur.

Chaque composant fournit ou requiert donc des services via une interface. Ces services ne doivent pas être perçus comme étant des fonctions mais plutôt comme étant des entrées ou sorties faisant partie d'un canal de communication. Ces services spécifient en fait des types d'objets de communication qui vont permettre à un composant d'utiliser une fonction d'un autre composant. Ces types d'objets sont gérés par un support d'exécution écrit en C++ nommé Regis. Le type d'objet le plus utilisé est le *port* et permet à un composant d'envoyer des requêtes vers d'autres composants. Nous complétons donc la déclaration d'un composant comme ceci :

```
component nomComposant (liste de paramètres) {
 provides nomPort <port, signature>
 requires nomPort <port, signature>
}
//où port désigne l'objet de communication de type port et
//signature désigne la signature de la fonction du composant.
```

Darwin ne possède pas la notion de connecteur mais utilise plutôt l'instruction **bind** qui relie le port requis d'une instance au port fourni d'une autre instance. Un second besoin important concernant les interconnexions entre les composants est de pouvoir vérifier leur compatibilité et interdire les interconnexions erronées (il s'agit de la propriété *safety*). C'est ainsi que l'opérateur bind ne permet pas la connexion entre deux interfaces non duales. Voici un exemple d'utilisation de l'instruction bind :

```
bind I1.A -- I2.B;
//où A et B sont deux interfaces duales.
```

Dans la suite de cette section, nous illustrons tous ces concepts à l'aide d'un exemple concret.

Exemple

Prenons l'exemple très répandu du *client-serveur*. Cet exemple décrit un composant de type *Application* comme une composition d'un client et d'un nombre paramétrable de serveurs.

Le nombre de serveurs est donc un paramètre du composant *Application* et exprime le nombre d'instances du composant de type *Serveur* dans une instance du composant de type *Application*. Les composants de type *Client* et *Serveur* s'expriment de la manière suivante :

```
component Client {
 requires requete <port, string>;
}
```

```
component Serveur {
 provides requete <port, string>;
}
```

Le composant *Application* s'exprime de la façon suivante :

```
component Application (int n) {
 Inst C : Client;
 Array S [0..(n-1)] : Serveur;
 forall k : 0..n-1 {
  inst S [k];
  bind S [k].requete   C.requete;
 }
}
```

Le mot clé **Array** permet de définir un tableau contenant un ensemble d'instances accessibles par un index. La clause **forall** permet de définir des instructions itératives. Illustrons désormais la clause *When* qui pose des conditions sur la manière dont les instances sont créées et liées entre elles. Imaginons que nous désirons lier le client uniquement aux serveurs occupant une position désignant un nombre pair dans le tableau. Voici la syntaxe qui serait utilisée dans ce cas bien précis :

```
when k%2=0
 inst S [k];
 bind S [k].requete   C.requete;

//où % représente l'opérateur modulo.
```

Avantages

Darwin est bien un langage de configuration et se distingue d'autres langages par sa capacité à instancier des types de composants. Ce mécanisme lui permet de décrire d'une manière précise les interactions entre les composants à l'exécution du système et de spécifier un schéma d'instanciation à l'initialisation de l'application. En effet, avec l'aide des clauses *forall* et *when*, il est possible de spécifier de manière dynamique la cardinalité des liens entre les différents composants.

Une autre originalité du langage est de fournir, en plus de l'instanciation par défaut, deux nouveaux modes d'instanciation : le mode paresseux ou dynamique. Par défaut, une instance est créée lorsque son instance parent est créée. L'instanciation paresseuse permet de créer une instance d'un type de composant seulement lorsque ses services sont requis par une autre instance. L'instanciation dynamique crée une nouvelle instance d'un type de composant a chaque appel d'une de ses fonctions.

De plus, Darwin fournit des mécanismes de description de la configuration physique d'un système. En effet, lors de la déclaration des instances, Darwin peut spécifier un numéro de site sur lequel cette instance peut être installée.

Finalement, darwin vérifie trois contraintes de sûreté au niveau des interactions entre les composants afin de réduire les erreurs susceptibles d'apparaître à l'exécution des applications. Premièrement, la vérification des contraintes de directionnalité interdit les liaisons entre deux interfaces non duales. Ensuite, la vérification des contraintes de connectivité empêche une instance de fournir une interface à plus d'une autre instance (fan out) mais autorise une instance à utiliser une interface fournie pas plusieurs autre instances (fan in). Enfin, Darwin possède aussi un système de vérification des types afin d'interdire les liaisons entre les interfaces de types incompatibles.

Inconvénients

Même si Darwin permet de configurer dynamiquement le schéma d'instanciation des composants à l'initialisation du système, il est incapable de modifier sa structure au cours du temps. Cela signifie qu'il est impossible de modifier des connexions, de voir apparaître ou disparaître des composants pendant l'exécution de l'application.

Ensuite, Darwin ne propose aucun mécanisme pour spécifier le caractère parallèle d'exécution des processus. Ce besoin doit être explicité par le programmeur.

Les communications multi-cast ne sont pas autorisées par le système de vérification des contraintes de connectivité. Or ce type de communication constitue un véritable atout pour certains systèmes distribués.

Parmi d'autres inconvénients de Darwin, nous pouvons citer le fait qu'il n'associe qu'une seule sémantique à la notion de composant. Il est donc impossible de spécifier des composants censés représenter d'autres éléments que le processus, comme des fichiers par exemple.

Finalement, le mécanisme permettant la spécification des communications inter composants est limité. En effet, l'utilisation de l'environnement Regis n'est pas transparent et les différents types d'objets de communications disponibles dans Darwin ne sont pas configurables.

1.2.2 Le modèle de composants Fractal

Le modèle de composants Fractal [6] est un modèle qui permet la description, la configuration et la reconfiguration dynamique d'une architecture d'un système à base de composants. Ce modèle correspond d'une part à une spécification et d'autre part à différentes implémentations dans plusieurs langages de programmation tels que Java, C ou C++. Fractal fournit un langage ADL proposant une syntaxe qui permet la construction des systèmes en respectant les spécifications du modèle. Fractal fut initialement conçu pour la construction, le déploiement et la configuration d'applications très complexes tels que les systèmes d'exploitation par exemple. Comme Darwin, Fractal est un modèle de composants qui utilise le mécanisme de composition afin de permettre aux développeurs de définir plusieurs niveaux d'abstractions. Par contre, Fractal se distingue dans sa capacité à partager des composants, à observer l'exécution d'un système, à proposer un modèle extensible ou encore à modifier une architecture dynamiquement pendant son exécution. Dans cette section, nous détaillons brièvement les concepts originaux qui font de Fractal un ADL puissant. Nous illustrons ensuite ces concepts à l'aide d'un petit exemple avant de conclure sur les avantages et les inconvénients.

Description de Fractal

Construire l'architecture d'un système avec Fractal revient à créer des composants et des liaisons entre eux afin qu'ils puissent communiquer. Un composant correspond à une unité d'exécution et possède une ou plusieurs interfaces. Chaque interface possède un type et implémente des opérations. Fractal distingue les interfaces *serveurs* (opérations fournies par un composant) des interfaces *clientes* (opérations requises par un composant). De plus, les interfaces sont soit *externes* (accessibles de l'extérieur du composant), soit *internes* (accessibles par les sous composants).

Un composant est constitué de deux parties :

* La **membrane** qui contient les interfaces du composant.
* Le **contenu** qui se compose d'un ensemble de sous-composants.

Dans la membrane d'un composant se trouvent les *contrôleurs*. Les contrôleurs sont en fait des objets qui se chargent de contrôler la configuration d'un système. Leur présence au sein de la membrane n'est pas obligatoire. Cependant leur combinaison permet de définir des contrôles avancés sur le comportement des composants et leurs interactions avec l'extérieur. Chaque sous-composant se trouvant dans le contenu est sous contrôle des contrôleurs présents dans sa membrane. Typiquement le contrôleur d'un composant composite peut :

* **Fournir une représentation** de la structure des interfaces des sous-composants.

* **Intercepter les invocations** d'opérations destinées ou venant des sous-composants.

* **Fournir un contrôle de gestion** du comportement supplémentaire à ceux fournis par les sous-composants, comme la suspension des activités d'un composant par exemple.

Il existe six types de contrôleurs différents :

* Le **contrôleur de liaisons** permet de créer ou détruire une connexion entre deux composants.

* Le **contrôleur d'attributs** permet de configurer les attributs d'un composant.

* Le **contrôleur de cycle de vie** permet de contrôler le comportement d'un composant pendant son éxecution. Par exemple, ce contrôleur se charge de créer ou détruire un composant dynamiquement. Ce dernier représente une caractéristique importante de ce langage car il permet la reconfiguration dynamique de composants à l'exécution du système.

* Le **contrôleur de contenu** ajoute ou supprime des sous-composants.

Comme tout langage de configuration, Fractal dispose de la notion d'instance. C'est ainsi que le Framework d'instanciation fût conçu spécialement pour permettre l'instanciation de composants. Dans ce modèle, les composants sont instanciés par d'autres composants, que l'on appelle *Factories*. Les Factories génériques peuvent créer plusieurs types de composants alors que les Factories standards ne peuvent créer que des instances d'un seul type.

Fractal fournit aussi un environnement d'introspection qui permet de parcourir par programmation l'ensemble des interfaces exposées par un composant ou encore l'ensemble des opérations spécifiées à l'intérieur d'une interface. Les interfaces d'introspection sont nécessaires afin d'obtenir la référence d'un objet dont on veut modifier son comportement.

Une autre originalité du langage Fractal réside dans le fait qu'il est possible de partager des composants. En effet, deux composants composites peuvent partager l'accès à un même sous composant qu'ils possèdent en commun. Ce mécanisme permet entre autres de modéliser l'accès à des ressources partagées.

Le système de type proposé par Fractal permet de déterminer la signature des fonctions définies dans une interface, de préciser si il s'agit d'interfaces *clientes* ou *serveurs*, ou encore de spécifier la caractère optionnel ou obligatoire d'une interface. Lorsqu'un composant possède une interface cliente et obligatoire, cela signifie que l'interface doit être liée pour que le composant puisse fonctionner. Par contre, les opérations d'une interface optionnelle ne sont pas nécessairement présentes et les composants qui la requièrent peuvent s'exécuter sans y être liés. Ce système permet aussi de spécifier les cardinalités d'une interface d'un certain type. Ces cardinalités déterminent le nombre d'interfaces qu'un composant peut posséder. Fractal distingue les interfaces *singleton* (une et une seule interface) des interfaces de type *collection* (ensemble d'un nombre quelconque d'interfaces). Aucune sémantique n'est imposée par Fractal en ce qui concerne l'exécution d'opérations des interfaces contenues dans une collection.

Comme Darwin, Fractal ne supporte pas la notion de connecteur. Il utilise plutôt le concept de liaison qui définit une connexion entre deux composants. Fractal vérifie aussi la contrainte de directionnalité d'une interaction entre deux composants. Le prochain paragraphe illustre à l'aide d'un exemple l'assemblage de composants pour former des configurations.

Exemple

La syntaxe de Fractal ADL se base sur XML pour spécifier les configurations de systèmes. Prenons l'exemple suivant tiré de [7]. Il décrit un composant composite contenant un client et un serveur connectés entre eux. Voici ce à quoi ressemble la description de cette architecture à l'aide de Fractal ADL :

```
<definition name="Application">
 <interface name="r" role="server" signature="java.lang.Runnable"/>
 <component name="Client">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <interface name="s" role="client" signature="Service"/>
  <content class="org.objectweb.julia.example.ClientImpl"/>
  <controller desc="primitive"/>
 </component>
 <component name="Server">
  <interface name="s" role="server" signature="Service"/>
  <content class="org.objectweb.julia.example.ServerImpl"/>
  <controller desc="primitive"/>
 </component>
 <binding client="this.r" server="Client.r"/>
 <binding client="Client.s" server="Server.s"/>
 <controller desc="composite"/>
</definition >
```

Avantages

Fractal est un modèle qui utilise des design patterns connus et les organise afin de former un modèle extensible et indépendant des langages de programmation. Comme la plupart des ADLs, il impose une séparation entre les aspects fonctionnels et les aspects de déploiement.

L' avantage principal que Fractal possède par rapport aux autres ADLs est sa capacité à reconfigurer dynamiquement un système pendant son exécution. Des composants peuvent changer de comportement, apparaître ou disparaître de la configuration du système. Dès lors, cela rend possible la construction de systèmes capables de gérer eux même leur propre cycle de vie. En effet, certains gros systèmes critiques (comme les gestionnaires de trafic aérien par exemple) ne peuvent se permettre d'utiliser une architecture uniquement statique.

Un autre avantage de Fractal est que son modèle de composant est simple et très léger. Les développeurs issus de la programmation orientée objet n'auront pas de difficultés à comprendre et utiliser ce modèle. De plus, une trace de l'architecture est toujours présente au niveau de l'implémentation, ce qui permet de faciliter la compréhension d'une architecture.

Comme nous l'avons signalé, Fractal ADL utilise XML pour assembler les composants et décrire les configurations de systèmes. La syntaxe utilisée par Fractal ADL est totalement extensible, ce qui signifie qu'il est possible de définir ses propres concepts architecturaux. De plus, XML est un standard très utilisé à l'heure actuelle et facile à prendre en main.

Finalement, la présence de contrôleurs au sein de la membrane permet d'exercer un certain pouvoir sur les différents éléments d'une architecture, ce qui permet la définition aisée de contraintes.

Inconvénients

Paradoxalement, les forces de Fractal ADL sont aussi des faiblesses. En effet, le principal inconvénient de Fractal est sa légèreté. De plus, Fractal ADL reste fort proche du modèle objet. Ces deux caractéristiques font de Fractal un langage assez pauvre et compliqué à analyser. Par exemple, analyser une liaison avec Fractal consiste uniquement à vérifier la signature des interfaces.

La syntaxe XML utilisée par Fractal est aussi un inconvénient dans le sens qu'elle rend plus difficile la lisibilité d'une architecture décrite. Il est nettement plus aisé de comprendre les concepts fondamentaux d'une architecture lorsqu'elle celle ci est décrite avec un langage de description plus spécifique, comme celui utilisé par Darwin par exemple.

Ensuite, contrairement à Darwin, Fractal n'introduit pas la notion d'instanciation au niveau de la définition d'une architecture. Il n'est donc pas possible de décrire, avec la même facilité que Darwin, le schéma de création et de connexion des instances à l'initialisation du système. Ceci est en partie lié à la syntaxe XML qui ne permet pas l'utilisation de clauses itératives (*forall* dans darwin) ou conditionnelles (*when* dans Darwin).

Finalement, son modèle d'interface est trop proche de celui utilisé par Java. Il distingue bien les interfaces fournies des interfaces requises mais une interface ne peut que fournir des services. C'est ainsi que pour modéliser avec Fractal une communication bidirectionnelle entre un client et un serveur, nous avons besoin de deux interfaces et de deux liaisons entre le client et le serveur. Fractal n'est donc pas adapté pour modéliser des services plus compliqués tels que nous en trouvons dans les réseaux par exemple où chaque composant possède la faculté d'initier une communication vers d'autres composants en utilisant un protocole identique.

Chapitre 2

Présentation du langage MADL

2.1 Introduction

Comme nous venons de le voir, plusieurs langages de description d'architecture ont été définis afin de décrire les architectures d'applications. Sur base de certains critères que nous avons présentés dans le précédent chapitre, ces langages peuvent être comparés et classés dans certaines catégories.

Cependant, la plupart de ces langages sont destinés à résoudre des problèmes spécifiques comme la décomposition d'une architecture par exemple (Darwin). Pour cela, ils se basent sur certaines hypothèses et par conséquent utilisent une sémantique qui limite leur utilité. C'est ainsi que certains langages, comme Darwin par exemple, ne permettent pas la distinction entre communications synchrones ou asynchrones au niveau de la définition de l'architecture. Par ailleurs d'autres langages imposent des architectures non distribuées ou statiques. En analysant la majorité des ADLs existants, nous pouvons remarquer que rares sont ceux capables de spécifier le cycle de vie d'un système et d'en modifier le comportement dynamiquement. Le but de MADL est de fournir un langage de description d'architectures capable de spécifier la structure d'une architecture à différents niveaux d'abstraction afin de permettre la définition d'un système à tous les stades de son cycle de vie. Un des avantages d'un tel langage est qu'il autorise les différents acteurs d'un projet à intervenir sur la même architecture, mais exprimée dans le langage familier d'un acteur en particulier.

Le langage de description d'architecture que nous définissons ici possède en fait plusieurs versions. Nous appelons MADL la syntaxe concrète textuelle du langage. Cette version utilise ce que l'on appelle des macro-commandes. Ces macro-commandes spécifient une architecture de manière plus condensée. Un exemple est l'instruction de boucle *for* que nous expliquons en détail dans la suite de ce chapitre. Concevoir une application revient donc à écrire une suite d'instructions appartenant à la syntaxe concrète du langage MADL. Dans ce chapitre, nous passons en revue cette version du langage et sa syntaxe.

La seconde version du langage est appelée ADL et peut être considérée comme un sous langage de MADL dont les macro-commandes ont été transformées en constructions équivalentes. Cette version est obtenue après compilation d'un programme écrit en MADL. Ce compilateur aplatit la structure d'une architecture. Cette version représente en fait l'*input* aux outils de génération de code pour un langage de programmation particulier.

Un métamodèle de description des concepts du langage MADL contient, après compilation d'un fichier MADL, les informations relatives à la structure d'une architecture ainsi que le schéma d'instanciation. Un autre métamodèle a été défini pour contenir les caractéristiques d'un réseau sur lequel un système décrit en MADL est susceptible d'être déployé. L'idée est de générer des informations de déploiement des composants sur un réseau à partir du croisement de ces deux métamodèles. Dans ce mémoire, nous considérons qu'un outil existe à cet effet.

L'objectif de ce chapitre est de présenter les concepts de base du langage ainsi qu'une sémantique pouvant lui être associée. Dans un premier temps et ce afin de faciliter la compréhension de MADL, nous exposons la sémantique la plus intuitive. Ensuite nous présentons brièvement le métamodèle. Enfin, nous illustrons l'usage de MADL à travers un exemple.

2.2 Concepts de base

Le langage MADL se base sur quatre éléments : le **type de composant**, l'**interface**, l'**ensemble d'instances** et le **connecteur**. Cette section a pour but de décrire les concepts de base du langage MADL et d'exposer leur sémantique. Pour chaque élément décrit, la syntaxe concrète du langage est détaillée et complétée par de petits exemples. Lorsqu'un élément fait partie de la description du métamodèle, le nom de la classe qui le représente est indiqué entre parenthèses et en italique.

2.2.1 Le type de composant (*Component_type*)

Le type de composant est une abstraction encapsulant un ensemble de propriétés et fonctionnalités utilisables par ses instances. En effet, comme nous le verrons dans les sections suivantes, MADL définit la notion d'ensemble d'instances qui sont en fait des matérialisations des types de composants.

La définition d'un type de composant peut contenir la définition d'autres constructions et est scindée en deux niveaux différents : la définition des **structures** d'une part et la définition des **configurations** d'autre part. Le premier niveau concerne la définition d'autres types de composants, de types d'interfaces et de contrats qui imposent des règles concernant leur utilisation. Le second niveau définit la déclaration d'ensembles d'instances, de connexions potentielles qui peuvent exister entre elles ainsi que le déploiement de ces instances.

La syntaxe qui permet de définir un type de composant en MADL est la suivante :

```
type_component nom\_du\_composant {
 // Définition de la structure du composant
 // Configuration du composant
}
```

Pour définir l'architecture d'une application à plusieurs niveaux d'abstraction différents, MADL utilise le mécanise de *composition*. La composition consiste à définir des types de composants capables de définir d'autres types de composants. L'architecture d'une application est en fait considérée comme une hiérarchie de types de composants où la racine représente l'application. Lorsqu'un type de composant comprend la définition d'autres composants, ce premier est appelé type de composant **composite**. Cette technique décompose un problème en sous-problèmes pouvant eux-même être décomposés en sous-problèmes jusqu'à la définition de types de composants **primitifs**. Un composant primitif ne contient aucune définition d'autres types de composants. Les instances de ces types de composants correspondent aux véritables unités d'exécution de l'application. L'instance d'un type de composant primitif utilise le terme *parent* pour désigner le type de composant composite dans lequel il est défini. Par contre, les instances des types de composants définis à l'intérieur d'un type de composant composite sont appelées *sous-composants*.

2.2.2 Le composant

Les composants sont typés, cela signifie qu'ils possèdent un type de composant. Un composant fait référence au déploiement de l'application alors que les types de composants représentent des "briques" permettant la construction de la structure d'une l'application. Le composant matérialise les fonctionnalités décrites par son type. On utilise aussi le terme d'*instance*.

Afin d'éviter toute confusion dans la suite de ce mémoire, il est important de fixer nos premiers mots de vocabulaire. Nous appelerons **instance**, tout composant capable de communiquer avec d'autres composants au déploiement et pendant l'exécution de l'application. Par contre, nous nommerons **variables**, les composants dont l'existence est prévue par le système mais qui n'ont pas encore été initialisés. Une variable est apparenté à un composant optionnel et ne peut communiquer avec l'extérieur.

La section 1.2.7 nous montre la cohérence de la distinction entre ces deux états.

2.2.3 Les interfaces (*Interface*)

Les interfaces représentent des points d'interactions entre les composants et forment ce que l'on appelle des protocoles. En MADL, une interface définit des **opérations** (*Operation*) ainsi que des **actions** (*Action*). Une opération représente un appel synchrone et bloquant pour l'appelant alors qu'une action correspond plutôt à un appel non bloquant et asynchrone. MADL étend le concept d'interface tel qu'il est défini dans la plupart des autres ADLs. En effet, une opération ou action est soit fournie ou requise par l'interface. Une opération fournie doit être implémentée par le composant qui expose l'interface. De façon duale, une opération requise est requise par ce composant. De cette manière, deux composants communiquant entre eux via un même protocole peuvent initier une communication.

La déclaration d'une interface utilise le mot clé *interface*, suivi d'un nom et de la définition des opérations et actions fournies ou requises. Les opérations et actions se définissent respectivement avec le mot clé *operation* et *action*. Ces opérations et actions peuvent recevoir et utiliser des paramètres (*Parameter*). Chaque paramètre possède un type et un type de passage. Une action étant un appel asynchrone non bloquant, elle ne peut renvoyer de valeur de retour. C'est pourquoi tous les paramètres d'une action représentent uniquement des paramètres d'entrée et aucun type de passage ne doit être spécifié. Par contre, une opération peut retourner une valeur vers l'appelant. C'est pourquoi le paramètre d'une opération peut être précédé des mots clés *in*, *out* ou *inout*. Finalement, comme nous le verrons dans la section suivante, une opération fournie (requise) est précédée du mot clé **prodives** (**requires**) alors qu'une action fournie (requise) est précédée du mot clé **in** (**out**). Les deux exemples suivant illustrent respectivement la définition d'une interface simple et bidirectionnelle :

```
interface BD {
 provides connect(in string login, in string mdp);
 in disconnect (string login);
}
```

Dans ce premier exemple, le composant qui fournit cette interface doit implémenter la méthode de connexion à la base de données et peut recevoir à tout moment une action de demande de déconnexion à cette base de donnée.

```
interface I {
 in action A;
 out action B;
 provides operation C;
 requires operation D;
}
```

Lorsque un type de composant fournit l'interface I (*provides I*), alors ses instances peuvent recevoir des événements de type A, et implémenter des opérations de type C. Par contre, elles peuvent émettre

des événements de type B et nécessitent des opérations de type D pour fonctionner correctement. Le cas du type de composant qui requière cette interface (*requires I*) représente le cas dual.

Les deux exemples montrent qu'une interface est soit fournie ou soit requise par un type de composant. Dans la prochaine section, nous allons voir qu'un type de composant requière ou fournit un type d'interface par l'intermédiaire de **contrats**.

2.2.4 Les contrats (*Contract*)

Les contrats représentent l'utilisation d'une interface par un type de composant et imposent des contraintes qui déterminent les trois caractéristiques d'une interface : la **catégorie**, la **cardinalité** et l'**obligation**.

Premièrement, nous pouvons distinguer deux catégories d'interfaces : les interfaces **serveurs** et **clientes**. Les interfaces serveurs (clientes) correspondent aux services fournis (requis) par le composant.

Deuxièmement, un contrat définit la **cardinalité** d'une interface, c'est à dire le nombre d'interfaces d'un certain type qu'un composant peut avoir. Lorsqu'un composant possède plusieurs interfaces de même type, nous utilisons le terme de *collection*. Dans le cas où un composant ne peut posséder qu'une seule interface, nous parlons d'interface *singleton*.

Finalement, un contrat spécifie si l'interface est **optionnelle** ou **obligatoire**. Une interface optionnelle ne doit pas être connectée pour que le composant fonctionne correctement. Par contre, l'interface obligatoire impose un nombre minimum de connexions à l'interface pour que le composant puisse fonctionner correctement.

La définition d'un contrat peut être décomposée en cinq éléments :

* un **mot clé** *requires* ou *provides* spécifiant la catégorie de l'interface.
* le nom désignant un **type d'interface**.
* **un alias** ayant pour but de renommer l'utilisation d'une interface. Un type d'interface peut ainsi être utilisé plusieurs fois mais pour des usages différents. Cet attribut est facultatif.
* **une cardinalité minimum** qui détermine si l'interface est optionelle ou obligatoire.
* **une cardinalité maximum** spécifiant le nombre maximum d'interfaces d'un même type que peut avoir un composant.

L'exemple suivant déclare un contrat dans le type de composant *Server* :

```
Component_type Server {
 provides 1,3 BD as bd;
}
```

Il s'agit ici d'une interface serveur : elle définit les services fournis par le type de composant (provides). De plus, cette interface est obligatoire. Dans ce cas ci, la cardinalité minimum indique qu'une de ses interfaces doit obligatoirement être connectée pour que le composant puisse fonctionner. Finalement, les instances du type de composant *Server* peuvent posséder un nombre maximum de trois interfaces de type *BD* (*collection*).

2.2.5 Les ensembles d'instances (*SetOfInstances*)

Une originalité du langage MADL réside dans la notion d'instance. Une instance est un composant conforme à un certain type. Alors qu'un type de composant est un concept abstrait n'intervenant qu'à l'étape de la conception d'une architecture, une instance correspond plutôt à une entité de déploiement capable de spécifier le comportement d'une application en implémentant les fonctionnalités décrites dans son type. En réalité, les types de composants ne sont rien de plus que des classificateurs ayant pour but d'encapsuler la définition de fonctionnalités et propriétés afin de faciliter la compréhension d'une architecture à sa conception. Les véritables composants d'un système sont représentés par les instances.

MADL déclare des instances en définissant ce que l'on appelle un *ensemble d'instances*. Cet ensemble regroupe un nombre fini de composants d'un même type. Un tel ensemble peut aussi bien contenir des composants à l'état d'instance qu'à l'état de variable.

Tout ensemble d'instances est borné par une cardinalité minimum et maximum. Formellement, un tel ensemble doit comprendre entre i et j instances opérationnelles où i et j sont respectivement les cardinalités minimum et maximum. Concrètement, i dénote le nombre de composants à l'état d'instance contenus dans l'ensemble. La différence entre j et i désigne le nombre de composants à l'état de variable. C'est ainsi que les ensembles d'instances dont la cardinalité minimum est plus grande ou égale à 1 introduisent la notion d'**instances obligatoires**. Une instance obligatoire est une instance qui doit être opérationnelle dès le déploiement du système afin que celui ci puisse fonctionner correctement.

La syntaxe utilisée par MADL pour déclarer un ensemble d'instances est la suivante :

```
deploy i-j nom_type_composant as nom_ensemble_instances ;
```

Chaque instance de cet ensemble est accessible de manière indicée. C'est ainsi que l'instruction nom_ensemble_instances[i], où i est un nombre plus grand ou égal à 0, permet d'accéder à l'élément se trouvant à la ième position de l'ensemble. Dans la prochaine section, nous verrons dans quel cas, cette instruction peut être utile. L'exemple ci-dessous définit un ensemble d'instances de type *Server* :

```
deploy 2-6 Server as servers;
```

Dans cet exemple, deux composants de cet ensemble sont des instances obligatoires. Les quatre autres composants ne seront pas déployés en tant qu'instance mais resteront à l'état de variable jusqu'à leur initialisation. La figure 2.1 schématise l'ensemble d'instances *servers* où les instances et variables sont respectivement représentées en vert et rouge.

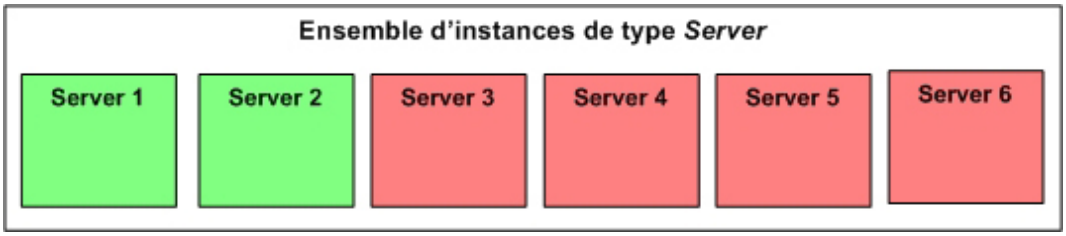

FIG. 2.1 – Représentation d'un ensemble d'instances de type Server.

2.2.6 Les connexions (*Connector*)

La connexion est ce qui permet aux instances de communiquer. Il existe deux types de connexions : **communication** et **délégation**. La sémantique d'une connexion réside uniquement dans la faculté qu'ont les composants à s'échanger des informations ou d'invoquer des services à distance. Une connexion ne désigne donc pas une communication "orientée connexion" comme nous pouvons en retrouver dans le protocole TCP par exemple. L'existence d'une connexion impose au pair initiateur de connaître, sous une forme quelconque, l'adresse de l'autre pair.

Connexion de communication (*Communication*)

La connexion de communication est établie entre une interface cliente et une interface serveur de type identique. Ainsi une instance a de type A peut être connectée à une instance b de type B, par l'intermédiaire d'interfaces duales de même type définies respectivement dans le type A et B.

Deux interfaces duales sont connectées par un **connecteur**. Un connecteur possède deux rôles distincts, que l'on nomme respectivement *in* et *out*. Le rôle *in* correspond à l'interface cliente alors que le rôle *out* correspond à l'interface serveur. Les connecteurs sont divisés en quatre groupes déterminant le mode de communication :

* **onetoone** : ce type de connecteur connecte une interface serveur à une interface cliente.

* **onetomany** : ce type de connecteur connecte une interface serveur à plusieurs interfaces clientes.

* **manytoone** : ce type de connecteur connecte plusieurs interfaces serveurs à une interface cliente.

* **manytomany** : ce type de connecteur connecte plusieurs interfaces serveurs à plusieurs interfaces clientes.

MADL est un langage abstrait et n'impose donc aucune sémantique particulière sur l'utilisation des connecteurs. Il ne précise pas comment les communications entre composants sont rendues opérationnelles. Le concepteur de l'application sera libre de définir, par la suite, la sémantique qu'il souhaite associer à un connecteur. Il pourrait par exemple s'agir des sémantiques suivantes :

* **Blocage des connexions interdites** : le connecteur refuse les connexions non autorisées par l'architecture. Seules les connexions définies à l'aide de connecteurs au niveau de la conception peuvent exister et être utilisées par les ensembles d'instances concernées. Par exemple, si aucun connecteur ne relie l'ensemble a à l'ensemble b, aucune instance de l'ensemble a ne peut communiquer avec une instance de l'ensemble b même si les interfaces déclarées dans leur type de composant sont duales et de même type.

* **Hétérogénéité de la communication** : le connecteur crée une communication entre deux instances de nature différente. C'est ainsi qu'une instance implémentée en java peut communiquer de manière transparente avec une instance implémentée en C++ par exemple. Le connecteur agit en fait de la même manière qu'un stub en Corba lorsqu'il emballe les requêtes et déballe les résultats.

* **Implémentation** : le connecteur peut aussi implémenter les exigences liées aux caractéristiques de déploiement (type de connexion et typologie physique). Exemple : la sécurisation d'une communication où le connecteur encrypte et décrypte les requêtes ou les résultats. Il joue dans ce cas le rôle d'agent de sécurisation de la communication.

Pour créer un nouveau connecteur, MADL fournit la syntaxe concrète suivante :

```
connector cardinalité_connecteur nom_connecteur ;
```

Ensuite ce nouveau connecteur peut connecter deux instances, via des interfaces duales et de même type, de la façon suivante :

```
connect instance1.nom_interface_cliente to nom_connecteur ;
connect nom_connecteur to instance2.nom_interface_serveur ;
```

L'utilisation combinée de ces deux instructions permet de spécifier des schémas de connectivité définissant les connexions potentielles qui peuvent exister entre les instances au déploiement d'une application. Les connexions qui ne sont pas explicitement déclarées dans ce schéma ne sont pas autorisées. L'exemple suivant définit une connexion *onetoone* entre une instance *c* de type *Client* et une instance *s* de type *Serveur*. Ces deux instances sont connectées par l'intermédiaire d'une interface *i* de type *I*. La figure 2.2 illustre la connexion de ces deux instances où les interfaces clientes et serveurs sont respectivement représentées en rouge et bleu.

```
connector onetoone communicationLink ;
connect c.i to communicationLink ;
connect communicationLink to s.i ;
```

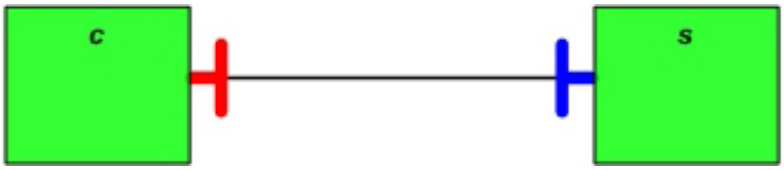

FIG. 2.2 – Définition d'une connexion de communication potentielle entre *c* et *s*.

Connexion de délégation (*Delegation*)

La connexion de délégation permet de déléguer l'implémentation d'interfaces supportées par un type de composants à d'autres types de composants. Un connecteur de délégation connecte l'interface d'un type de composant composite à l'interface d'un de ses sous-types de composants. Les deux interfaces connectées doivent posséder la même déclinaison ainsi que le même type. L'invocation d'un service sur l'interface du composite sera transmise au sous composant relié par délégation.

La syntaxe qui définit ce type de connecteur est légèrement différente et utilise le mot clé supplémentaire *forward* :

```
forward connector nom_connecteur;
```

Voici un exemple qui définit un connecteur de délégation entre une instance *s* de type *Server* et une instance *w* de type *WebService*. Les deux instances sont reliées par leur interface serveur *r* de type *Request*.

```
forward connector delegationLink;
connect s.r to delegationLink;
connect delegationLink to w.r;
```

Lorsqu'un service de l'interface *r* fournie par *s* (l'opération *getTime()* sur la figure 2.3) est invoquée par une instance (*c* par exemple), l'invocation de service est transmise à l'instance *w*. L'instance *c* ne sait pas qu'elle est en fait en train de dialoguer avec *w*.

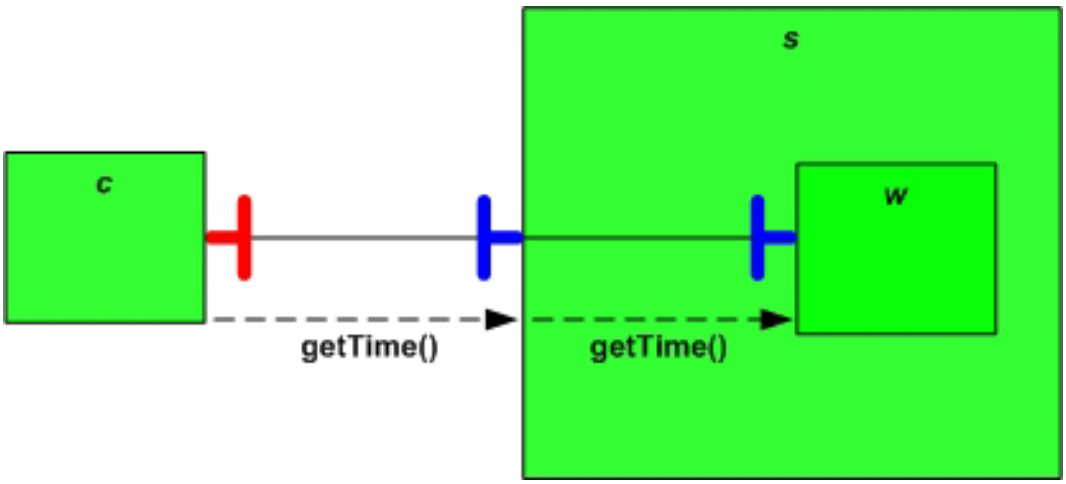

FIG. 2.3 – Définition d'une connexion potentielle de délégation entre s et w.

2.2.7 La configuration

MADL permet de définir la configuration de déploiement d'une application en combinant l'utilisation des trois mécanismes que nous venons de montrer, à savoir la *composition*, l'*instanciation* et la *connexion*.

Une configuration est perçue par MADL comme un espace dans lequel sont disposés des emplacements instanciables (*instanciation*) ainsi que des liens qui les connectent ensemble (*connexion*). Un emplacement instanciable est représenté par un composant se trouvant à l'état de variable. Chaque emplacement peut en contenir d'autres (*composition*). Lorsqu'une application est déployée, sa configuration définit un ensemble de variables initialisées, c'est à dire des composants à l'état d'instance, qui représentent les composants obligatoires. Ces composants sont nécessaires au système pour fonctionner correctement. Les autres composants restent à l'état de variable mais pourront, en cas de besoin, être transformés en instances durant l'exécution de l'application.

Nous pouvons en fait créer une analogie avec une carte mère. La carte mère est munie de *slots* pouvant accueillir plusieurs types de cartes et composants hardware (disques durs, carte sons, cartes graphiques, mémoire RAM, etc...). Ces composants peuvent communiquer par l'intermédiaire de bus de communication et la présence de certains est obligatoire afin de faire fonctionner un ordinateur. C'est le cas par exemple de la mémoire RAM qui est dans ce cas un composant obligatoire. Ensuite il est possible d'ajouter de nouveaux composants permettant ainsi d'ajouter de nouvelles fonctionnalités à un ordinateur.

La figure 2.4 décrit la configuration d'un système défini par composition des types de composants *Application*, *Client* et *Server*. *Client* et *Server* sont des sous-types de composants de *Application*. Le système ne peut pas fonctionner sans la présence d'une instance de type *Server* et *Application*. Ces instances sont représentées en vert. Les autres composants représentent des emplacements instanciables (variables). Leur initialisation en instance peut avoir lieu plus tard durant l'exécution de l'application. Des connexions potentielles sont décrites entre l'instance de *Server* et les variables de type *Client*.

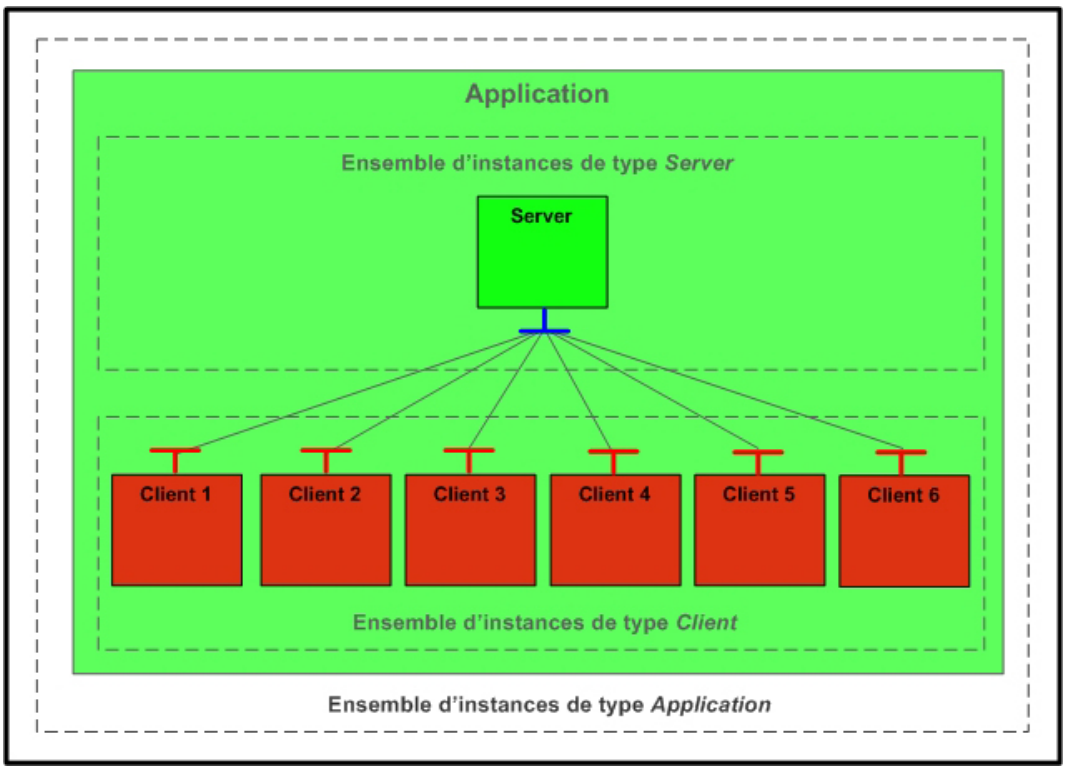

FIG. 2.4 – Définition de la configuration d'une application.

Pour faciliter la construction de la configuration de déploiement, MADL fournit l'instruction *For*. Cette construction de boucle permet de spécifier l'architecture d'un système. Il est ainsi possible de condenser la déclaration des connexions entre instances.

L'exemple ci-dessous montre l'utilisation de l'instruction *For* dans le cadre de la connexion des deux ensembles d'instances *Servers* et *Clients*.

```
deploy 1-1 Server as servers;
deploy 0-10 Client as clients;
connector onetomany connecteur;
connect servers.BD to connecteur;
for (int i=0 ;i<10 ;i++) {
 connect connecteur to clients[i].BD ;
}
```

2.2.8 La paramétrisation d'architecture

Il n'est pas toujours souhaitable de connaître à l'avance le nombre d'instances qu'un ensemble peut contenir. C'est pourquoi MADL introduit le concept de paramétrisation d'un type de composant. Ce mécanisme permet à un type de composant de se configurer en fonction de données d'entrée qui ne sont connues qu'au déploiement du système. Les paramètres d'un type de composant sont compris entre "<" et ">" et suivent immédiatement le nom du type de composant.

Cet exemple montre l'utilisation d'un tel paramètre pour fixer le nombre maximum de clients que l'application peut contenir :

```
type_component Application <int M> {

 type_component Client {
  ...
```

```
 }
 deploy 1-M Client as clients;
}
```

2.2.9 Attributs d'architecture (*Attribute*)

Dans [8], les auteurs montrent la pertinence d'attacher des attributs à des parties de l'architecture pour permettre la direction de transformations et de raffinages de l'architecture. L'originalité du langage est de permettre à des acteurs différents de travailler sur la même architecture mais à des niveaux d'abstraction différents. C'est ainsi que quatre niveaux d'abstractions sont identifiés : le niveau conceptuel, logique, physique et d'infrastructure. Chaque niveau correspond donc à une phase du projet dirigé par un acteur différent.

Un attribut représente soit des propriétés (besoins fonctionnels et non fonctionnels) ou soit des caractéristiques. Ces dernières dénotent des propriétés qui sont remplies par les structures de l'architecture. Le principe du raffinement d'architecture sur base d'attributs est de transformer les propriétés liées à un niveau en caractéristiques. En d'autres mots, nous pouvons considérer qu'un type d'architecture se trouve au niveau logique lorsque toutes les propriétés liées au niveau conceptuel sont devenues des caractéristiques du système et qu'il ne reste plus que des propriétés logiques.

Voici quelques exemples d'attributs qu'il est possible d'attacher à une architecture :

* **cohesion** : la cohésion dénote la nature du composant. Des valeurs possibles sont *distributed*, *host*, *process* ou *thread*.
* **moveable** : la capacité qu'a un composant à changer d'host durant l'execution. Trois valeurs sont possibles : *yes*, *no* ou *unknown*.
* **memory** : cet attribut précise le type de mémoire lié à un composant (*stateful*, *stateless*, *memoryless* ou *unknown*).
* **parallelism** : la valeur *parralel* signifie qu'un composent peut exécuter des invocations en parallèles sur son interface. Mis à *serial*, le composant sérialise les demandes d'accès à l'interface.
* **secure** : cet attribut indique si la confidentialité des données échangées au sein d'une connexion est assurée ou pas.
* **critical** : cet attribut indique que le composant est crucial au bon fonctionnement du système dans lequel il est déployé.

Le principe consiste donc à transformer toutes ces propriétés en structures d'architecture qui implémentent les besoins liés à ces propriétés. Les nouvelles structures sont ensuite annotées avec les caractéristiques résultantes. Une méthode classique consiste à transformer un type de composant annoté avec des attributs en sous composants qui ont pour rôle d'implémenter les attributs liés au type de composant parent.

2.3 Le métamodèle

Le métamodèle définit un ensemble de classes représentant les structures du langage MADL ainsi que les relations qui existent entre elles. Il sera utilisé comme référentiel pour contenir l'information propre à une description d'architecture en vue de développer un outil de génération de code.

La figure 2.5 nous montre l'ensemble de ce métamodèle sous forme de diagramme de classe décrit dans le langage UML[9].

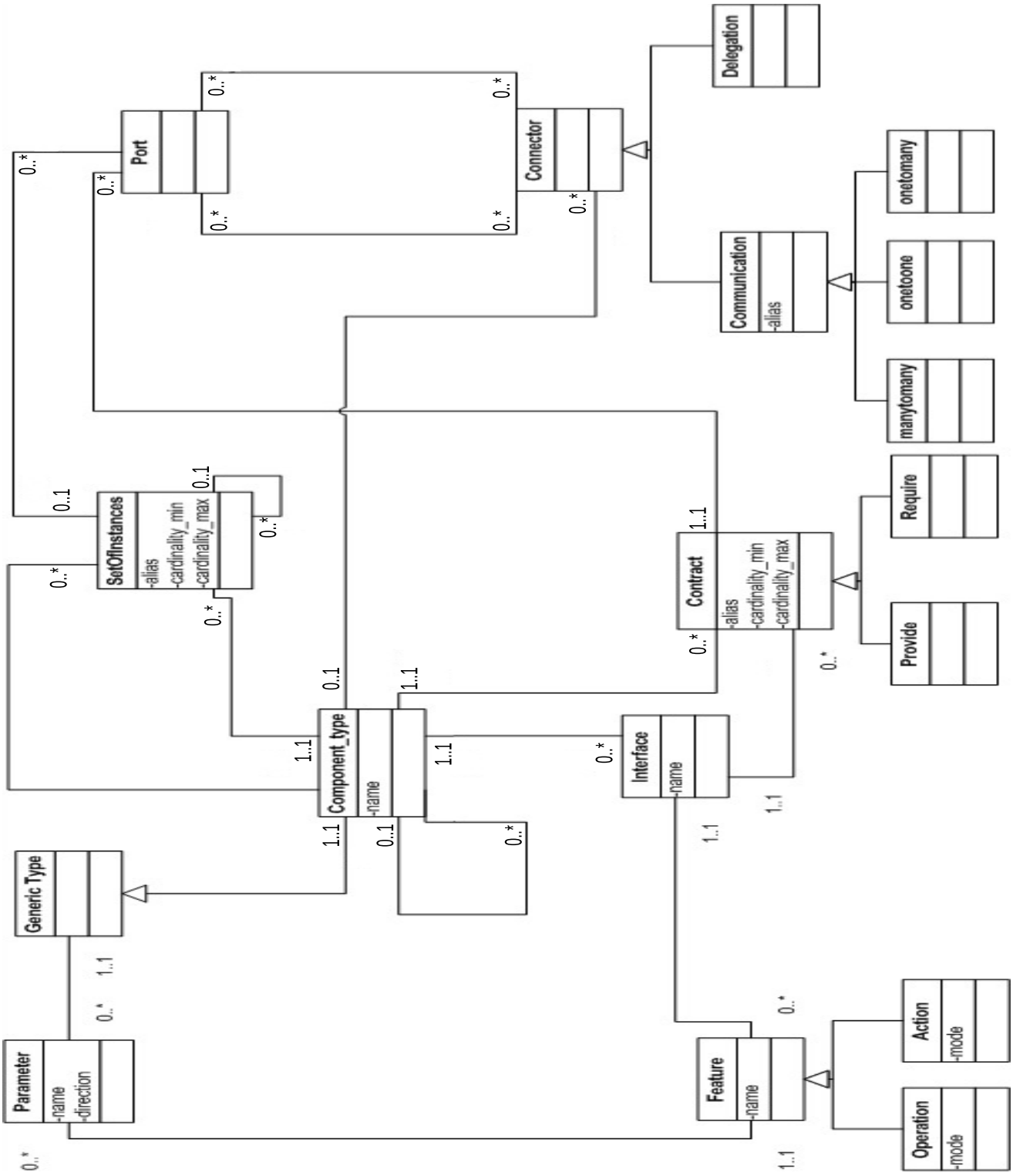

FIG. 2.5 – Description du métamodèle en UML.

2.4 Un exemple

Finissons ce chapitre en illustrant les concepts de base de MADL par un exemple simple. Il s'agit ici de construire une application de type *client-serveur*. Le serveur fournit une interface par laquelle plusieurs clients peuvent communiquer en appelant à distance la méthode *reverse*.

```
component_type Application <int N> {

 interface Business {
  provides operation string reverse (string chaine);
 }

 component_type Client {
  requires 1-1 Business;
 }

 component_type Server {
  provides 1-5 Business;
 }
 deploy 0-N Client as clients;
 deploy 1-1 Server as servers;
 connector one_to_many link;
 connect servers.Business to link ;
 for (int i=0 ; i<=N ; i++) {
  connect link to clients[i].Business ;
 }
}
deploy 1-1 Application < 2 > as root.
```

L'application décrite ci dessus correspond à un type de composant composite paramétré qui contient deux types de composant primitifs. Sa définition se décompose en deux niveaux : la définition des structures d'une part et la définition de la configuration du type de composant racine d'autre part. En effet, comme nous pouvons le constater sur l'exemple ci-dessus, les types d'interfaces et de composants sont tout d'abord définis. Le type de composant *Application* définit une interface de type *Business* qui fournit une méthode *reverse()* exécutant une opération sur une chaîne de caractères et renvoyant comme résultat une autre chaîne de caractères. L'application définit en second lieu les deux types de composants primitifs *Client* et *Server*. Le type de composant *Client* définit un contrat de type *require* imposant aux futures instances de créer au minimum une connexion à une interface de type *Business* pour pouvoir fonctionner correctement. Un autre contrat est défini dans le type de composant *Server* et spécifie que ses instances doivent fournir au minimum un point de connexion en permanence à l'interface *Business* et qu'elles ne peuvent supporter plus de 5 connexions simultanées.

Le second niveau concerne la déclaration des ensembles d'instances et des connexions potentielles qui peuvent exister entre elles. Une instance du type de composant *Server* est déclarée comme obligatoire dans le sens où sa présence est nécessaire dès le déploiement de l'application. Un ensemble d'instances de n clients est déclaré et aucune de ses instances n'est obligatoire. Le nombre n est un entier et représente un paramètre du type de composant *Application*. L'instruction *For* permet de connecter l'interface cliente avec l'interface serveur via un connecteur nommé *link*. Ce connecteur dénote une communication *onetomany* : un serveur communique avec plusieurs clients dans le cadre de l'utilisation de l'interface *Business*.

Finalement l'application est déployée et le paramètre N est fixé. Le type de composant *Application* représente la racine de la hiérarchie, c'est pourquoi nous nommons son instance *root*.

Chapitre 3

Cas d'étude : Mapping vers Corba IDL

3.1 Introduction

Le *Mapping* est un terme anglais très souvent utilisé dans le domaine de l'informatique et plus particulièrement dans le développement de langages de programmation. Il désigne l'action de mettre en correspondance deux éléments distincts. Par exemple, la spécification de Corba dispose de règles de Mapping mettant en correspondance les types de variable IDL avec les types de variables Java.

L'objectif des ADLs est de permettre la conception d'un système informatique en utilisant une syntaxe concrète spécifique afin de spécifier la structure de son architecture ainsi que son comportement. Une architecture décrite dans son niveau d'abstraction le plus détaillé est enfin prête à être déployée sur de véritables supports physiques, à savoir des ordinateurs. Pour cela, les ADLs disposent de règles de mapping faisant correspondre les structures du langage en d'autres structures équivalentes appartenant à des langages de programmation spécifiques. Par exemple, Darwin dispose de règles permettant à une description d'architecture d'être déployée sur des ordinateurs à l'aide du langage orienté objet Java.

Le but de ce chapitre est de valider certains choix conceptuels intervenant dans la définition du langage MADL à travers un cas d'étude consistant à établir des solutions de mapping vers le middleware Corba. Dans un premier temps, nous présentons brièvement le middleware Corba. Nous exposerons ensuite nos solutions de mapping. Ces solutions correspondent à la sémantique la plus intuitive de MADL, comme nous l'avons vu lors du précédent chapitre.

3.2 Présentation de Corba

CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture) [10] est une norme créée en 1992 par l'*OMG* (**O**bject **M**anagement **G**roup), organisation regroupant des fournisseurs de logiciels et des utilisateurs finaux. Cette norme est une spécification permettant de construire des applications distribuées à base d'objets communiquant ensemble. Ces derniers pouvant être écrits dans des langages de programmation différents ou encore être déployés sur des machines distinctes et cela de manière transparente pour les développeurs participant à la conception d'une application. L'élément clé de communication entre ces objets se nomme l'*ORB* (**O**bject **R**equest **B**roker) et fera l'objet d'une description plus détaillée dans la suite de cette section. Ensuite nous nous attarderons brièvement sur les *ObjectServices* qui sont en fait des services de base permettant la gestion des Objets Corba. Nous terminons par une présentation du processus de développement d'une application Corba. Cette section présente uniquement une vue d'ensemble du middleware Corba. Le lecteur intéressé désirant en savoir plus peut se rapporter aux spécifications de l'OMG [10].

3.2.1 L'ORB

L'ORB est l'élément central qui fournit les mécanismes fournissant aux objets la capacité d'envoyer des requêtes et de recevoir des résultats de manière transparente. Il garantit l'interopérabilité entre des applications distribuées sur des machines distinctes au sein d'environnements hétérogènes.

Corba utilise le terme *servant* pour désigner l'implantation des objets Corba. Grâce à l'ORB, des applications *serveurs* initialisent ces servants en les préparant à recevoir des requêtes. Le rôle principal de l'ORB est de fournir aux applications *clientes* des mécanismes de localisation des servants, et d'assurer ensuite la transmission des requêtes.

Une application client doit connaître la référence d'un serveur avant de pouvoir envoyer une requête aux servants qu'il gère. Au préalable, cette application cliente doit aussi connaître la signature de la méthode qu'il désire appeler. Chaque servant est représenté du coté client par un *stub* dont son rôle est la génération de requêtes. L'ORB transfert ensuite la requête à l'objet implémentant le serveur via un *skeleton*. L'objet distribué exécute la requête et renvoie les résultats.

L'ORB comprend les composants suivants : IDL (**I**nterface **D**efinition **L**anguage), DII (**D**ynamic **I**nvocation **I**nterface), IR (**I**nterface **R**epository), OA (**O**bject **A**dapter).

Le langage de définition d'interface (IDL)

Rendre des objets hétérogènes interopérables entre eux nécessite une définition de leur interface indépendamment de la définition des caractéristiques d'implémentation. Cela s'accomplit à l'aide du langage IDL.

IDL est un langage qui permet la définition du type des objets en spécifiant leur interface. L'interface d'un objet représente un ensemble d'opérations comportant des paramètres.

Comme nous venons de le signaler, un client doit connaître le type de l'objet et les signatures des opérations qu'il fournit avant de pouvoir lui envoyer une requête. Ceci est accompli grâce aux interfaces IDL qui sont accessibles au client via un entrepôt d'interfaces IDL (Interface Repository).

Lorsque ces interfaces sont compilées, les *stubs* des clients et les *skeletons* des serveurs sont générés.

Interface d'invocation dynamique (DII)

L'invocation statique n'est pas capable de localiser de nouveaux objets ajoutés plus tard au système. L'invocation dynamique résout ce problème en permettant à un client à l'exécution d'accéder à de nouveaux objets, de découvrir leur interface et de construire dynamiquement des requêtes prêtes à être envoyées à ces nouveaux objets.

L'invocation dynamique utilise des *stubs* génériques capables d'envoyer n'importe quelle requête vers n'importe quel objet. Cependant ce type d'invocation, faisant appel à certains services de l'ORB, peut s'avérer assez coûteuse en temps d'exécution.

L'entrepôt d'interfaces (IR)

L'IR est le composant de l'ORB qui contient toutes les définitions d'interfaces des objets composant le système et qui gère leur accès. Ces interfaces sont principalement utilisées par l'ORB afin de vérifier le type des signatures des requêtes.

L'adaptateur d'objet (OA)

Les adaptateurs d'objets sont des objets responsables d'invoquer les requêtes sur des implémentations de serveurs (appelés aussi *servants*) via leur *skeleton*. En général, un adaptateur d'objets gère plusieurs servants selon une politique de gestion spécifique. Lorsqu'ils reçoivent une requête émanant d'un client, ils sélectionnent tout d'abord le servant correspondant et invoquent ensuite la méthode désirée à travers un *skeleton*.

Les adaptateurs d'objets sont aussi responsables de la génération des références d'objets (appelées aussi IOR) et de leur gestion. De plus, ils offrent des mécanismes permettant l'activation et la désactivation de servants.

3.2.2 ObjectServices

L'ORB fournit un ensemble de services de base mettant à disposition des applications des mécanismes de gestion des objets Corba. Les trois services les plus connus sont le *service de nommage* (Naming Service), *service de gestion des évènements* (Event Service) et le *service de gestion du cycle de vie* (Lifecycle Service).

Le service de nommage

Le service de nommage est un mécanisme utilisé par les objets d'un système afin de localiser les objets dont ils ont besoin. Un service de nommage consiste en un répertoire de noms où le nom représente une séquence de composants. Chaque composant est constitué de deux attributs : un identifiant (*identifier*) et un type de composant (*kind*). Chacun des composants (à l'exception du dernier) fait référence à un contexte. A l'intérieur d'un contexte, chaque nom est unique.

Dans ce répertoire, chaque objet est donc lié à un nom et enregistré dans un contexte particulier. Le but du service de nommage est de déterminer l'objet associé à un nom donné pour un contexte donné. Cette action s'appelle *résolution* de nom.

Le service de gestion des évènements

Le service de gestion des évènements est un mécanisme fourni par l'ORB qui permet l'envoi de messages asynchrones entre objets distribués d'un système. Il utilise le pattern *publish/subscribe*.

Ce pattern définit la notion de producteurs et de consommateurs. Les producteurs déclenchent des évènements et les consommateurs consomment les données liées à ces évènements. Dans le modèle *push*, le producteur initie le transfert de données vers les producteurs alors que dans le modèle *pull*, c'est le consommateur qui possède l'initiative en demandant le déclenchement d'évènements au près des producteurs.

Les producteurs et consommateurs communiquent via l'utilisation d'un objet CORBA standard : un *event channel*. Cet event channel représente l'intermédiaire et permet de relier plusieurs producteurs avec plusieurs consommateurs.

Le service de gestion du cycle de vie

Le service de gestion du cycle de vie permet de créer, détruire ou déplacer des objets. La création d'objets s'exécute par l'intermédiaire d'une *factory*. Une factory est objet qui crée d'autres objets. Ces factories ne sont pas des objets à part et possèdent comme les autres objets une interface décrite en IDL. L'interface LifeCycleObject définit les opérations de destruction, copie et déplacement d'objets.

3.2.3 Processus de développement d'une application Corba

Une application Corba se compose d'un ensemble d'objets distribués communiquant ensemble par l'intermédiaire d'interfaces définissant les services qu'ils rendent. Ces objets sont lancés par une application serveur. Ce serveur a pour but d'initialiser le composant ORB et d'attendre les requêtes qui sont ensuite distribuées au servant via un *skeleton*. Des applications clientes localisent ces objets afin d'utiliser leur service via un *stub*. Une application peut agir en tant que serveur et client en même temps.

Voici les étapes retraçant le processus de développement d'une telle application :

1. **Définition des interfaces IDL** : les services rendus par les objets du système sont définis à travers une interface IDL. Cette étape est indispensable et constitue l'input de l'étape suivante.
2. **Compilation des interfaces IDL** : Un compilateur prend en input les interfaces IDL et génère des classes Java correspondant aux *stubs* des clients, et aux *skeletons* des serveurs. Des règles de mapping assurent la transformation des types IDL en types Java.
3. **Implémentation des servants** : cette étape consiste à fournir une implémentation des services rendus par les servants. Il s'agit du code business. Ce code détermine le comportement réel des objets distribués.
4. **Définition des conventions de nommage** : Les applications clientes localisent les objets corba via le service de nommage (*naming service*). Ces objets sont enregistrés sous un nom. Afin de faciliter la localisation de ces objets par les clients, des conventions de nommage sont décidées au préalable.
5. **Initialisation des clients et des serveurs** : Cette dernière phase consiste à écrire le code qui a pour but d'initialiser l'ORB des clients et des serveurs. Ensuite, les classes java correspondant aux applications clientes et serveurs sont prêtes à être déployées sur des machines. Enfin, le *naming service* est lancé et les serveurs peuvent s'y enregistrer.

3.3 Définition de solutions de mapping

MADL est un langage qui se trouve encore au stade de sa définition. Nous avons présenté dans le chapitre précédent une des sémantiques possibles liée aux différents concepts de base du langage. Cette sémantique est la plus intuitive. Ce chapitre n'a pas pour but de définir des règles précises de Mapping vers Corba mais plutôt de considérer ce cas d'étude comme un procédé de validation des différents choix conceptuels qui ont été effectués dans la définition du langage. De plus, ce chapitre va élaborer des pistes de solutions non encore prises en compte dans la conception du langage MADL et ayant pour but de définir des mécanismes qui vont permettre aux structures d'une architecture MADL d'être interconnectées et déployées au sein d'un système réparti.

Dans un premier temps, et pour un souci de facilité de compréhension, nous considérons uniquement les interfaces unidirectionnelles et les connecteurs simples. Les interfaces bidirectionnelles et les autres types de connecteurs seront traités ultérieurement. De plus, nous ne prenons pas en compte l'évolution dynamique d'un système au cours de son exécution. Cette partie fera l'objet d'une réflexion dans un chapitre ultérieur.

Ce cas d'étude et les solutions de mapping que nous exposons ici se basent sur l'exemple du *client-serveur* détaillé à la fin du chapitre 2. Même si les solutions élaborées peuvent être considérées indépendantes d'une technologie particulière, nous avons choisi CORBA IDL comme support technologique afin d'illustrer et valider ces règles de transformation.

La définition de ce mapping se base sur les principes du processus détaillé à la fin de la section précédente. Nous allons donc commencer par définir les interfaces IDL définissant les attributs et méthodes

associés aux concepts de base du langage. Ensuite, nous émettons une solution de localisation des composants afin d'autoriser leurs interconnexions. Nous présenterons ensuite les mécanismes de création des variables et d'initialisation des composants d'un système. Finalement, nous détaillons le comportement des composants leur permettant d'envoyer et recevoir des requêtes.

Une dernière section sera consacrée au mapping de deux constructions plus particulières : les interfaces bidirectionnelles et les connexions de délégation.

3.3.1 Définition des interfaces IDL

L'intégration des concepts de base du langage MADL au sein d'un modèle de programmation comme Corba nécessite dans un premier temps l'apport de règles de définition d'interfaces IDL. Ces règles vont permettre la définition des interfaces des trois structures de base du langage MADL. Ces structures doivent être matérialisées en objets Corba pour permettre à une architecture MADL d'être déployée dans un environnement distribué selon les spécifications du middleware Corba IDL.

Dans cette section, nous allons donc définir des règles pour les trois structures de base du langage, à savoir le *type de composant*, l'*interface* et le *connecteur*.

Le type de composant

Les types de composants sont les éléments clés d'une architecture décrite en MADL. Par mécanisme de composition hiérarchique, ils forment sa structure. Si nous jetons un regard sur le middleware Corba, un seul élément est susceptible de représenter l'équivalent d'un type de composant. Cet élément est l'**interface IDL**. En effet, le type de composant et l'interface IDL représentent tous deux des abstractions de concepts pouvant être à la base de la formation d'un système. Le type de composant est une abstraction d'un composant alors que l'interface IDL représente l'abstraction d'un objet. Le composant compose la structure de l'architecture d'un système à composants alors que l'objet compose celle d'un système à objets. Nous pouvons donc remarquer une certaine équivalence entre la notion de type de composant dans MADL et la notion d'interface IDL dans Corba.

Dès lors, chaque type de composant est transformé en une interface IDL portant le même nom, et contenant des propriétés et des méthodes spécifiques ayant pour but de matérialiser son rôle au sein du système. Sachant qu'un composant a pour rôle principal l'initialisation de ses sous-composants et la création de ses connecteurs, l'interface IDL d'un type de composants définit uniquement deux **opérations d'initialisation**, *initSubComponents()* et *initConnectors()*. Ces deux méthodes autoriseront, au déploiement, les composants à passer de l'état de variable à l'état d'instance. Nous étudierons leur fonctionnement plus tard dans ce chapitre.

Si un type de composant équivaut à la définition d'une interface IDL, dès lors un composant (ou instance d'un type de composant) équivaut à un objet Corba. Cet objet implémente les services définis dans l'interface IDL du type de composant. En effet, un composant en MADL matérialise un certain type de composant. En Corba, un servant matérialise les opérations contenues dans une interface IDL. La figure 3.1 illustre la correspondance qui existe entre d'une part le type de composant et l'interface IDL, et d'autre part entre le composant et l'Objet Corba :

Les interfaces unidirectionnelles

Une interface unidirectionnelle fournit des opérations ou peut recevoir des événements. Contrairement aux interfaces bidirectionnelles, elles ne peuvent pas requérir d'autres opérations ou déclencher d'autres évènements. Chaque interface MADL est **transformée en interface IDL**. Toutes les opérations et actions décrites au sein de l'interface unidirectionnelle sont transformées en méthodes IDL qui

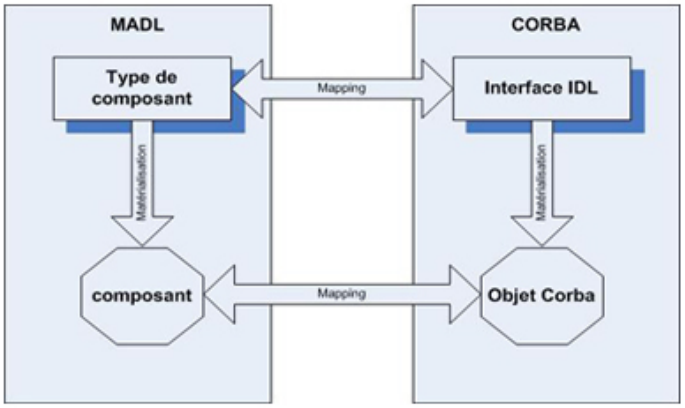

FIG. 3.1 – Correspondance entre le type de composant et l'interface IDL.

seront, au déploiement du système, implémentées par un objet Corba, aussi appelé **objet d'interface**.

Par défaut, les opérations contenues dans une interface IDL dénotent des appels de méthode synchrones et bloquants. Chaque opération MADL équivaut donc à une **simple opération IDL**. En ce qui concerne la transformation des actions MADL, CORBA dispose de deux mécanismes différents où chacun possède des avantages et inconvénients.

Le premier mécanisme, le plus trivial, revient à déclarer une opération IDL avec le **mot clé oneway**. Ce mot clé spécifie que lorsqu'un objet invoque cette opération, aucune réponse n'est attendue ou reçue. La sémantique de l'invocation d'une opération *oneway* est *best-effort*, ce qui ne garantit pas la délivrance de l'appel. Une opération *oneway* ne doit pas posséder de paramètres de retour et doit être définie avec le type de retour *void*. Considérons l'interface MADL ci-dessous :

```
interface Exemple1 {
 provides operation string methodeA (string parametre);
 in action actionA (string parametre);
}
```

Si nous appliquons nos règles de transformation sur cette interface unidirectionnelle, nous obtenons l'interface IDL suivante :

```
interface Exemple1 {
 string methodeA (in string parametre);
 oneway void actionA (in string parametre);
};
```

Notons que la spécification IDL exige l'utilisation du mot clé **in** précédant chaque paramètre d'entrée.

La deuxième solution consiste à utiliser des **event services**. Dans ce cas, les composants qui implémentent (requièrent) une action deviennent des consommateurs (producteurs) d'événements. Contrairement aux opérations *oneway*, la sémantique du déclenchement d'un évènement n'est pas best-effort. Tous les évènements produits sont assurés d'être consommés. Cependant, un tel procédé fait référence à l'utilisation d'*event channels*, ce qui complexifie la mise en place d'un système.

Le tableau suivant compare l'utilisation des deux mécanismes permettant de transformer une action MADL en structures Corba.

	Opérations *oneway*	*Events services*
Inconvénients	Best effort	Complexe à mettre en place
Avantages	Facile à implémenter	Assurance de délivrance des évènements

FIG. 3.2 – Récapitulatif des deux procédés.

Les connecteurs

Conceptuellement parlant, un connecteur est perçu comme une entité entière établissant une connexion potentielle entre deux ensembles d'instances. Cependant, les connecteurs possèdent deux rôles distincts dans une communication, appelés rôles *in* et *out*. Cette distinction des rôles chez un connecteur nous amène à le **distribuer** en deux sous-ensembles. A partir du connecteur initial, nous obtenons donc d'une part les **connecteurs in** qui correspondent au rôle in et d'autre part les **connecteurs out** correspondant au rôle out.

Tout composant relié par un connecteur dispose d'un des deux types de connecteurs selon qu'il fournit ou requière un certain type d'interface. Si un composant requière un certain type d'interface, il disposera d'un connecteur in, responsable d'envoyer des requêtes et de recevoir des résultats. Par contre, lorsque ce composant fournit une interface, un connecteur out est créé, le rendant capable de recevoir des requêtes et de transmettre des résultats. Sur la figure 3.3, nous montrons un connecteur, distribué en un connecteur in et trois connecteur out. Ce connecteur relie une instance *A* à trois instances *B1*, *B2* et *B3*. La couleur verte (rouge) désigne le rôle in (out).

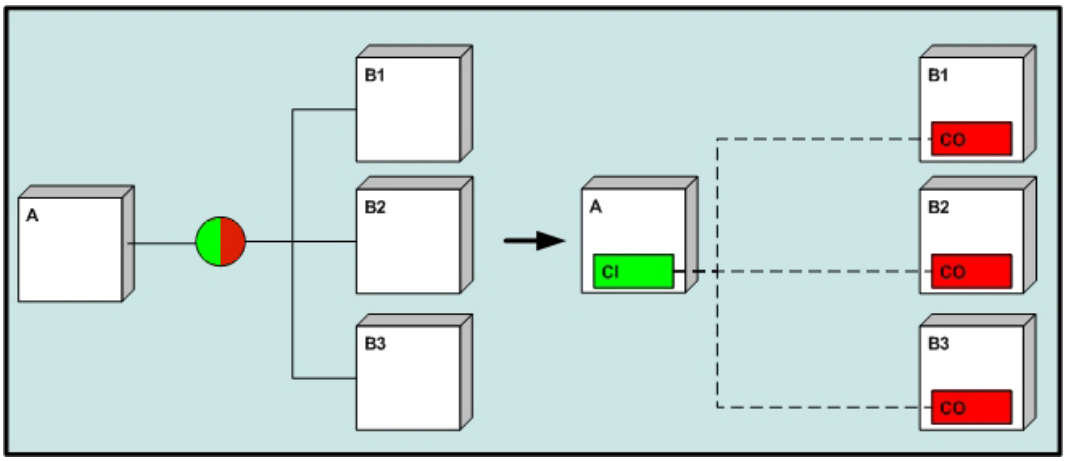

FIG. 3.3 – Distribution d'un connecteur en connecteurs in et out.

Si nous nous référons à l'exemple ci-dessus, il est tout à fait envisageable que *CI* soit déployé sur un autre support que *CO*. Or, comme nous le verrons dans la section 3.3.3, l'instance *A* peut envoyer des messages vers l'instance *B1* si et seulement si *CI* détient la référence de *CO*. Dans ce cas, *CI* et *CO* sont matérialisés par des objets Corba distribués proposant des services de connexion et d'utilisation d'interfaces accessibles à distance.

Tout connecteur est donc converti en interface IDL (portant le nom du connecteur) qui définit des attributs et des méthodes ayant pour objectif la matérialisation du rôle in (out) pour un connecteur in (out).

L'interface IDL d'un connecteur **in** définit :

* un **ensemble de connecteurs out** impliqués dans la même connexion. Ces références sont ob-

tenues à l'initialisation du connecteur in et utilisées ensuite pour utiliser les services exposés par un composant.

* des **méthodes simples** et **oneway** dont leur signature correspond respectivement aux opérations et actions définies dans le type interface faisant l'objet de la connexion. Ces méthodes et actions vont permettre à l'instance qui requière une interface d'appeler les services fournis.

* une **méthode d'initialisation**. Cette méthode sera responsable de l'obtention de la référence des connecteurs out impliqués dans la connexion.

L'interface IDL d'un connecteur **out** définit :

* un **ensemble d'objets d'interfaces** implémentant le type d'interface dont la connexion fait l'objet. Ces références sont créées à l'initialisation du connecteur out et utilisées pour traiter les requêtes.

* des **méthodes simples** et **oneway** dont leur signature correspond respectivement aux opérations et actions définies dans le type interface faisant l'objet de la connexion. Ces méthodes et actions vont permettre la transmission des requêtes vers l'objet d'interface correspondant.

* une **méthode d'initialisation** responsable de l'enregistrement du connecteur out dans le naming service.

Reprenons l'exemple illustré sur la figure 3.3. Ci-dessous se trouve la syntaxe concrète définie par le langage MADL pour connecter l'interface *require* de l'instance A à l'interface *provide* des instances *B1*, *B2* et *B3*. Le type d'interface concerné par la connexion est *Exemple1*. En appliquant les nouvelles règles de production d'interfaces IDL pour les connecteurs, nous obtenons deux interfaces IDL nommées respectivement *linkIn* et *linkOut*.

```
connector onetomany link ;
connect A.Exemple1 to link ;
connect link to B1.Exemple1 ;
connect link to B2.Exemple1 ;
connect link to B3.Exemple1 ;
```

L'interface IDL de *linkIn* définit un tableau d'éléments ou chacun d'entre eux est une référence vers un connecteur de type *linkOut*, les méthodes *methodeA* et *actionA* décrites dans le type d'interface *Exemple1* et une méthode d'initialisation du connecteur.

```
interface linkIn {
 // les connecteurs out
 typedef linkOut ConnectorOut[] ;
 attribute ConnectorOut connectorOut ;
 // les méthodes simples et oneway
 string methodeA(in string parameter) ;
 oneway void actionA(in string parameter) ;
 // la méthode d initialisation
 void init() ;
};
```

L'interface IDL de *linkOut* définit un tableau d'éléments ou chacun d'entre eux est une référence vers un objet d'interface de type *Exemple1*, les méthodes *methodeA* et *actionA* décrites dans le type d'interface *Exemple1* et une méthode d'initialisation du connecteur.

```
interface linkOut {
 // les objets d interface
 struct SetOfExemple1 {
 Exemple1 exemple1 ;
 boolean status ; };
 typedef SetOfExemple1 Interfaces[] ;
 attribute Interfaces interfaces ;
 // les methodes simples et one way
 string methodeA(in string parameter) ;
 oneway void actionA(in string parameter) ;
 // la méthode d initialisation
 void init(); };
```

3.3.2 Localisation des composants

A ce stade, nous savons désormais que les types de composants, les interfaces unidirectionnelles et les connecteurs possèdent un équivalent en terme d'interfaces IDL où chacune d'entre elles définissent un ensemble d'attributs et de méthodes. Nous venons aussi de constater que les types de composants sont susceptibles d'être hébergés par des machines différentes et que leur interconnexion est assurée par la distribution de connecteurs in et out. Cependant, une question importante se pose : comment les composants qui requièrent une interface peuvent-ils localiser les composants qui la fournissent ? La réponse à cette question se trouve dans l'utilisation d'**un naming service** et la définition de certaines **conventions de nommage**.

Le naming service des connecteurs

L'échange de messages entre deux composants nécessite dans un premier temps leur interconnexion. Or, des composants ne peuvent pas être connectés directement entre eux mais utilisent pour cela des connecteurs in et out. Dans le cadre d'une communication unidirectionnelle, un composant possède un connecteur in lui permettant d'envoyer des messages qui seront interceptés par d'autres composants grâce à un connecteur out. Nous dirons dans ce cas qu'un composant A est connecté à un composant B lorsque le connecteur in de A détient la référence du connecteur out de B. Nous rappelons que l'utilisation de la référence d'un d'objet distribué est indispensable afin d'appeler à distance les services qu'il rend.

Les connecteurs in obtiennent la référence des connecteurs out par l'intermédiaire du **naming service des connecteurs**. Nous avons montré qu'un composant rend accessibles les services qu'il rend à travers l'utilisation de connecteurs out. Lorsque ce composant est déployé, il enregistre la référence de ces connecteurs out dans le service de nommage. Ainsi, leur référence devient disponible et utilisable par n'importe quel composant. La figure 3.4 illustre l'utilisation du naming service par le connecteur in du composant A désirant localiser la référence du connecteur out du composant B.

Fig. 3.4 – Utilisation du naming service des connecteurs

Sur ce schéma, les connecteurs in et out sont représentés respectivement en vert et rouge. Premièrement, le composant B enregistre la référence de son connecteur out dans le *naming service*. Ensuite, le

connecteur in peut résoudre cette référence grâce à laquelle le composant *A* sera capable d'appeler les services rendus par *B*.

Un *naming service* est un référentiel d'objets accessibles par leur **nom** (*name*). Ce nom est utilisé par les connecteurs in afin d'obtenir la référence d'un connecteur out enregistré dans ce référentiel. Cela signifie qu'un connecteur in a uniquement besoin de connaître le nom sous lequel la référence d'un connecteur out est enregistré afin de pouvoir l'utiliser. Le *naming service* est structuré en différents contextes de noms (*naming contexts*) qui désignent des sous-répertoires dans lesquels les objets peuvent enregistrer leur référence. Définir plusieurs *naming contexts* à l'intérieur d'un *naming service* autorise une meilleure organisation des associations "nom-reference".

Chaque nom est structuré de manière hiérarchique où chaque élément de son arborescence est appelé composant de nom (*name component*) et est constituée de deux membres, le nom (*id*) et genre (*kind*). De manière analogique, nous pouvons comparer la structure de ce nom aux noms de fichiers dans un système de fichiers. Pratiquement, un nom est composé d'une séquence de *name component* où chaque *name component* désigne soit un *naming context* ou soit une référence CORBA. Chaque *name component* d'un nom se termine par une barre oblique et un point sépare les deux membres *id* et *kind*. L'attribut *kind* d'un *name component* permet la distinction des noms enregistrés dans un certain *naming context* et possédant une valeur identique pour l'attribut *id*.

Ainsi, l'exemple suivant représente un nom composé de trois *name component* où les deux premiers désignent un *naming context* et le dernier désigne une référence CORBA :

```
Nourriture.dir/Fruits.dir/Pomme.obj
```

La figure 3.5 décrit un naming service composé de plusieurs *naming contexts* contenant la référence de deux connecteurs out. Les noms *connecteur1* et *connecteur2* proviennent de la description d'une architecture en MADL.

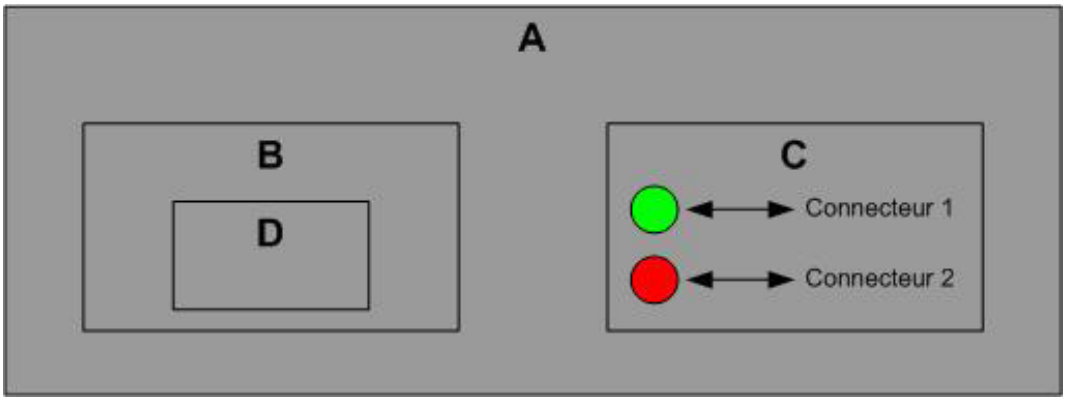

FIG. 3.5 – Utilisation du naming service des connecteurs

Les références de ces connecteurs out ont été enregistrées dans le *naming context* décrit par la séquence suivante :

```
A.comp/C.comp
```

Nous avons choisi l'extension "*comp*" pour l'attribut *kind* de chaque *name component* afin de montrer que chaque *naming context* provient de la définition d'un type de composant en MADL. Un connecteur in peut obtenir la référence de *connecteur1* par exemple en utilisant le *name component* suivant :

```
A.comp/C.comp/connecteur1.co
```

Par exemple, l'extension "*co*" est utilisée pour montrer que *connecteur1* est l'*id* d'un connecteur out.

Les conventions de nommage

Les connecteurs in et out ont besoin d'établir des conventions de nommage communes en ce qui concerne l'enregistrement des références d'objets dans le *naming service*. En effet, le connecteur in doit connaître la manière dont un connecteur out enregistre sa référence afin de la résoudre pour une utilisation ultérieure. Ces conventions vont s'accorder sur deux informations : **le lieu d'enregistrement** des références et **la structure du name component** propre à cette référence.

Le *naming service* des connecteurs est organisé en hierarchies de *naming contexts*. Chaque *naming context* désigne l'espace occupé par un type de composant provenant d'une description d'architecture MADL. C'est ainsi qu'une architecture définissant un type de composant A qui lui même définit deux sous types de composants B et C, donnera lieu à la création de trois *naming contexts* nommés NA, NB et NC où NB et NC sont des *naming contexts* contenus dans le *naming context* principal NA. La référence d'un connecteur out est enregistrée au sein du *naming context* correspondant au type de composant qui a défini ce connecteur. Par exemple, si A définit un connecteur nommé *link*, sa référence est enregistrée dans le *naming context* désigné par *NA.comp/*.

Il nous reste à fixer les conventions concernant la valeur que prendront les attributs *id* et *kind* des *name components* correspondant aux références des connecteurs out. L'attribut *id* prendra comme valeur le nom du connecteur concaténé à la chaîne de caractère "out" alors que l'attribut *kind* contiendra un numéro unique visant à distinguer tous les connecteurs out de même type enregistrés dans le même *naming context*. Effectivement, un connecteur *onetomany* implique l'enregistrement de plusieurs connecteurs out de même type et au sein du même *naming context*. L'exemple ci-dessous montre la structure du *name component* associée à la référence d'un connecteur out appelé *link* :

```
linkOut.3
```

L'extension "3" signifie qu'il existe trois connecteurs out différents mais de type identique enregistrés dans le même *naming context*.

3.3.3 Initialisation des composants

Chaque composant possède deux états possibles. Lors de sa création, il se trouve dans un premier état que nous qualifions de **variable**. Dans cet état, le composant existe mais ne peut échanger de messages avec d'autres composants car il n'a pas encore créé les objets qui vont lui permettre de communiquer avec l'extérieur. Pour passer dans un second état, que nous appelons **instance**, le composant doit exécuter une opération d'initialisation. Le schéma ci-dessous nous montre le passage d'un composant à l'état de variable (en noir) vers l'état d'instance (en rouge). Lorsqu'il se trouve dans l'état d'instance, un composant est capable d'envoyer et de recevoir des messages.

FIG. 3.6 – Initialisation d'un composant.

Pendant cette phase d'initialisation, un composant :

* crée les **connecteurs in et out**
* **initialise** les connecteurs
* crée et initialise ses sous-composants

Chaque composant implémente les deux opérations suivantes : *initSubComponents()* et *initConnectors()*.

Création des connecteurs

Les composants qui requièrent une interface unidirectionnelle créent un connecteur in alors que ceux qui la fournissent créent un connecteur out. Ce sont ces connecteurs qui interconnectent les composants leur permettant alors d'échanger des messages.

La figure 3.7 définit le processus de création des connecteurs in et out, créés respectivement par un composant de type *A* et *B*.

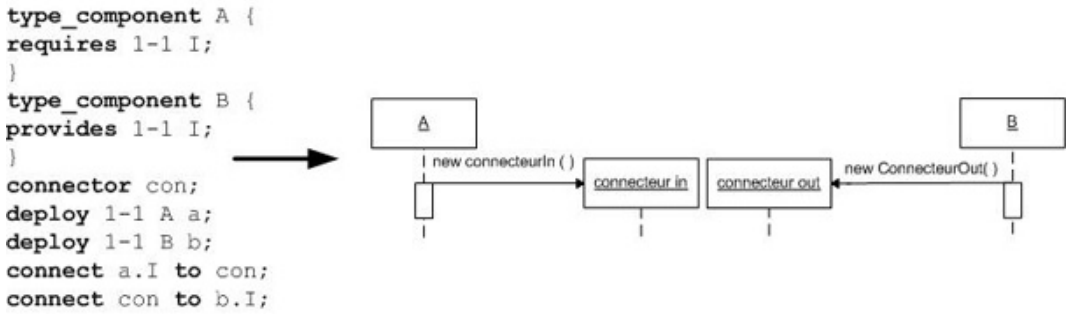

FIG. 3.7 – Création des connecteurs in et out

Initialisation des connecteurs

Lorsqu'un composant à créé un connecteur, il procède ensuite à son initialisation, phase durant laquelle le connecteur met en place les mécanismes lui permettant d'établir les connexions entre les composants. L'initialisation d'un connecteur dépend de son type. Tout type de connecteur implémente la méthode **void** init() qui est définie dans son interface IDL.

Un connecteur in (out) obtient dans un premier temps la référence du *naming service* à partir de laquelle le connecteur in résout la référence des connecteurs out alors que les connecteurs out y enregistrent leur propre référence. Les connecteurs out créent aussi les objets implémentant le type d'interface connectée. Ce processus est détaillé dans les quatre paragraphes suivants.

Localisation du Naming Service Le naming service est un objet distribué dont sa référence peut être obtenue vie une instance de l'ORB. N'importe quelle application peut obtenir cette référence en fournissant l'adresse IP de la machine qui héberge le service et le port l'exécutant. CORBA fournit le protocole standard *corbaloc* pour obtenir les références des objets générés par l'ORB. C'est ainsi que l'url suivante permet d'obtenir la référence d'un *naming service* exécuté sur la machine 192.21.3.4 et sur le port 2001 :

```
corbaloc:iiop:1.2@192.21.3.4:2001/NameService
```

Afin de rendre la localisation du *naming service* transparente, chaque connecteur utilise un résolveur d'adresse. Le résolveur est en fait une entité connaissant les informations de localisation du *naming*

service et qui est capable de fournir ces informations selon un protocole spécifique, comme UDP par exemple. De cette façon, le *naming service* peut être délocalisé sans affecter l'exécution des connecteurs.

Résolution des références Le naming service fournit des services d'exploration de son arborescence. Ainsi il est possible, depuis n'importe quelle application distante, de lister le contenu d'un *naming context* particulier.

A son initialisation, le connecteur in obtient dans un premier temps la référence du *naming context* désignant le type de composant qui a défini ce connecteur. Son contenu est ensuite listé et les références qui y sont enregistrées sont résolues une à une par le connecteur in. Le nombre de références résolues à l'initialisation d'un connecteur in correspond à la cardinalité minimale du contrat de type *require*.

Supposons qu'une instance *A1* de type *A* soit connectée à trois instances de type *B*, nommées *B1*, *B2* et *B3* par un connecteur *onetomany* appelé *link*. Supposons aussi que le type de composant *A* définit un contrat de type *require* spécifiant une cardinalité minimale égale à deux. Dans ce cas, à son initialisation, l'instance *A1* se positionne sur le *naming context* dans lequel elle est enregistrée et résout, au choix, deux des trois références.

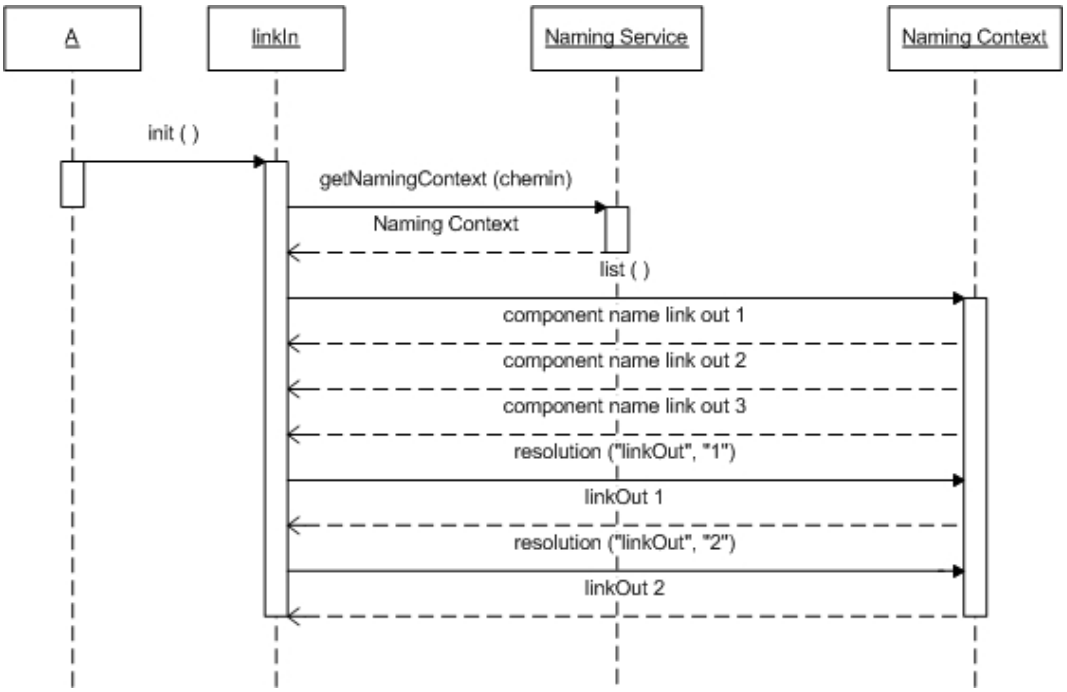

FIG. 3.8 – Résolution de la référence de deux connecteurs out.

Enregistrement des références Le connecteur out s'enregistre dans le *naming context* adéquat et s'assure que la valeur de son attribut *kind* soit supérieure à la plus grande valeur d'attribut *kind* déjà contenue dans le *naming context*. Ainsi, tous les connecteurs out de même type enregistrent leur référence sous un *component name* différent, ce qui permet leur différentiation par les connecteurs in.

Création des objets d'interface Les opérations et actions définies dans les interfaces unidirectionnelles sont implémentées par des objets d'interfaces. Les composants qui fournissent une interface unidirectionnelle ne créent pas directement les objets d'interfaces correspondants. C'est à l'initialisation du connecteur out que ces objets sont effectivement instantiés.

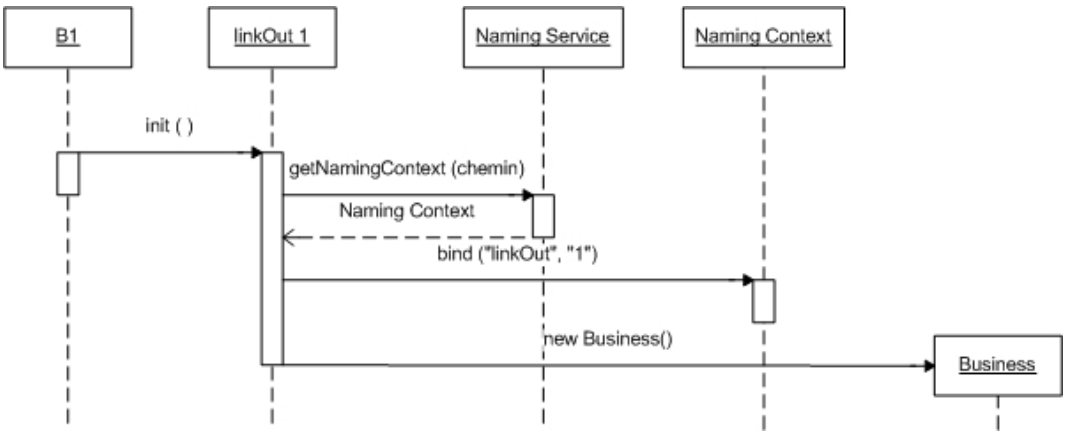

FIG. 3.9 – Enregistrement de la référence d'un connecteur out.

Ce sont les contrats de type *provide* qui spécifient le nombre d'objets d'interface effectivement créés lors de l'initialisation du connecteur out.

Sur la figure 3.10, un composant de type B initialise un connecteur out nommé *linkOut*. Ce dernier crée deux objets d'interfaces de type I car B définit un contrat de type *provide* spécifiant que le composant doit obligatoirement fournir deux possibilités de connexion en permanence.

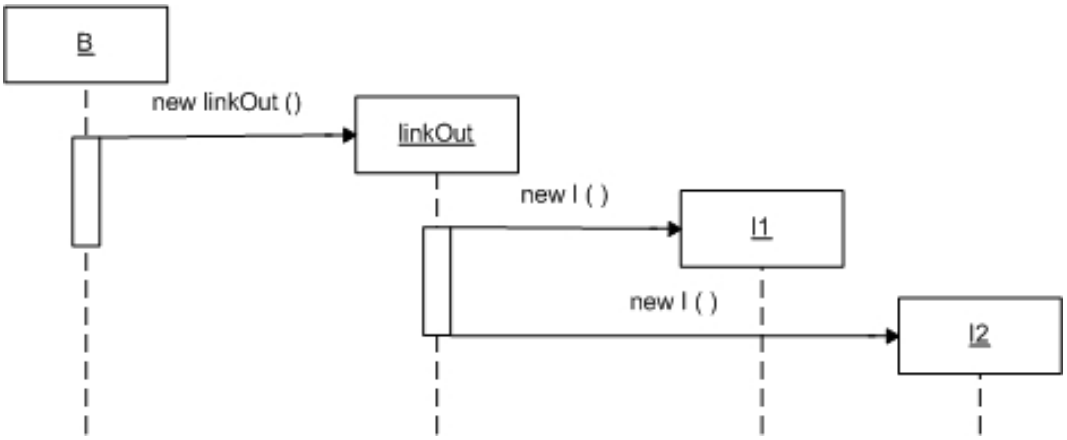

FIG. 3.10 – Création des objets d'interfaces d'un composant.

Sur les trois schémas précédents, nous supposons que le *naming service* a préalablement été localisé par l'intermédiaire du résolveur.

Création et Initialisation des sous-composants

Le passage d'un composant, d'un certain type, à l'état d'instance se termine par la création et l'initialisation des sous-composants définis dans ce type. Cette nouvelle phase est conditionnée par les ensembles d'instances déclarés. Comme nous allons le voir dans la suite de cette section, ce processus est utilisé afin de déployer les composants obligatoires d'une architecture de manière récursive.

En conclusion, nous avons montré qu'un composant ne possède finalement pas d'autre rôle que d'initialiser son contexte lui permettant d'émettre et de recevoir des requêtes. La gestion de la connexion avec les autres composants est déléguée aux connecteurs alors que l'implémentation des interfaces est déléguée aux objets d'interfaces.

3.3.4 Déploiement du système

MADL décrit un système comme un ensemble de variables et de connexions qui peuvent exister entre elles. Une fois initialisées, ces variables contiennent des instances capables de fournir des services à d'autres instances ou d'en utiliser. Certaines variables doivent être initialisées dès le déploiement de l'application pour qu'elle puisse fonctionner correctement. Par contre, d'autres sont optionnelles à ce stade du cycle de vie du système et seront initialisées par la suite.

Le déploiement des composants est le processus permettant de reproduire la configuration d'un système, conformément à la description d'un schéma d'instanciation, en créant les variables et les premières instances nécessaires à son fonctionnement. Nous présentons ici les principes utilisés par ce processus.

Les factories

[11] définit un ensemble de solutions aux problèmes apparaissant de manière récurrente dans la conception d'applications. Il existe parmi les patterns responsables de l'initialisation de classes (patterns de création), celui de la *Factory*. Une factory est un objet responsable de la création d'autres objets. Il existe des factories standards responsables de la création d'un seul type d'objet et les factories génériques capables de créer des instances de types différents.

Des factories standards sont utilisées afin de construire l'ensemble des variables qui vont constituer le système. Chaque factory est responsable de la création de variables d'un **certain type** sur **un support physique** particulier. Ces factories sont des objets Corba qui permettent de créer des variables à distance et de renvoyer ensuite leur référence. Ainsi une application externe est capable d'utiliser les services de création des factories afin d'initialiser tout un système. La figure 3.11 montre un exemple d'utilisation de ces factories dans le cadre d'une application définissant trois variables de type *Client* déployées sur un support physique nommé *machine 1* et une variable de type *Server* déployée sur un support physique nommé *machine 2*.

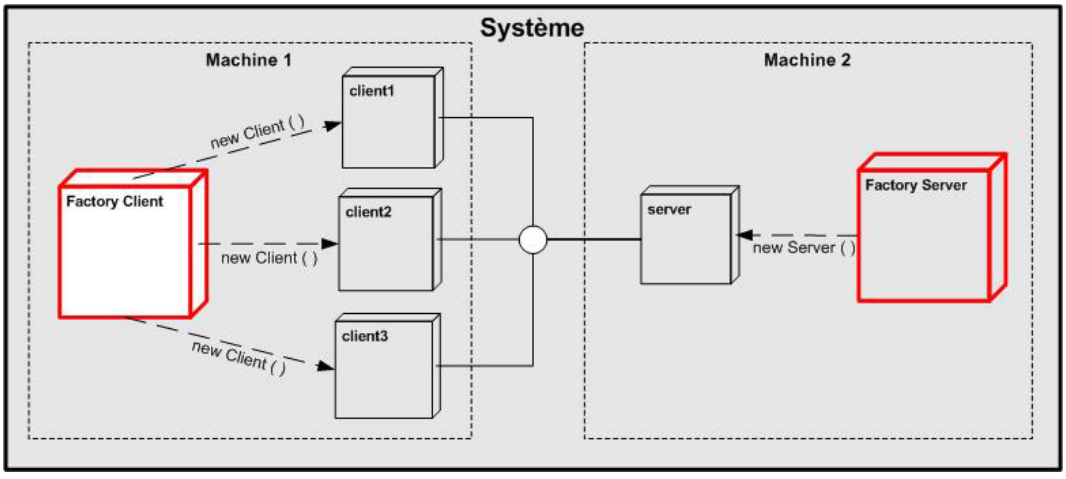

FIG. 3.11 – Utilisation de factories standards

Déploiement des composants

Les composants d'un système sont déployés de manière récursive. Cela signifie que chaque composant composite déploie les sous-composants qu'il a défini. Ce processus de déploiement est repété récursivement jusqu'au moment où tous les composants ont été créés. Dans un premier temps, les composants sont déployés à l'**état de variable**. Ils sont créés grâce à une factory standard.

Illustrons par un exemple (figure 3.12), le déploiement de trois composants A, B et C où A définit B et B définit C. Le composant C est un composant primitif. Les flèches rouges indiquent les appels aux services de création des factories alors que les flèches noires dénotent des créations d'objets. Finalement, *initialisation* représente un composant externe qui a pour rôle d'amorcer le processus de déploiement.

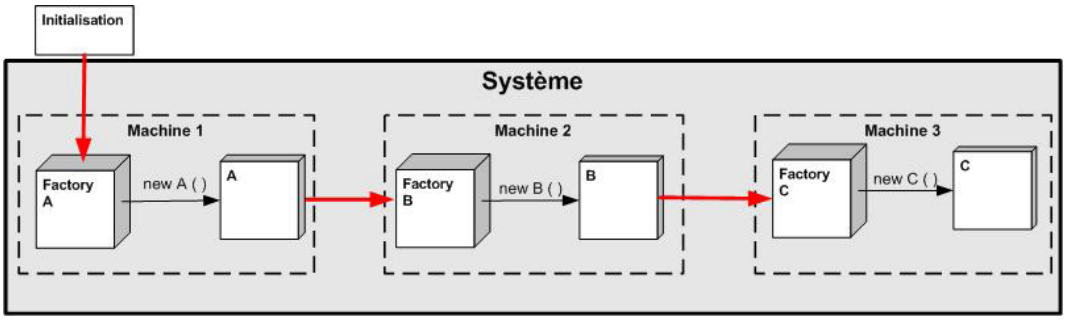

FIG. 3.12 – Déploiement des variables par principe de récursivité.

Pour refléter les contraintes de déploiement du système, les composants obligatoires passent ensuite de l'état de variable à l'état d'instance. Cette transition s'effectue par exécution de deux méthodes d'initialisation, *initSubComponents ()* et *initConnectors()*, fournies par chacun des composants. Cette méthode peut être éxecutée car chaque composite reçoit de la factory la référence du composant créé.

Utilisons un autre exemple (figure 3.13) pour montrer l'initialisation d'un composant se trouvant à l'état de variable. Ici, le système comporte la définition de trois composants A,B et C où A définit B et C, et où A et B représentent deux composants obligatoires. Premièrement, sous l'impulsion d'un composite, le service de création de la factory est appelé afin de créer la variable correspondante (flèches rouges). Ensuite, la factory retourne la référence de cette variable au composite (flèches vertes claires). Finalement, le composite utilise cette référence pour initialiser le composant qui devient désormais une instance (flèches vertes foncées).

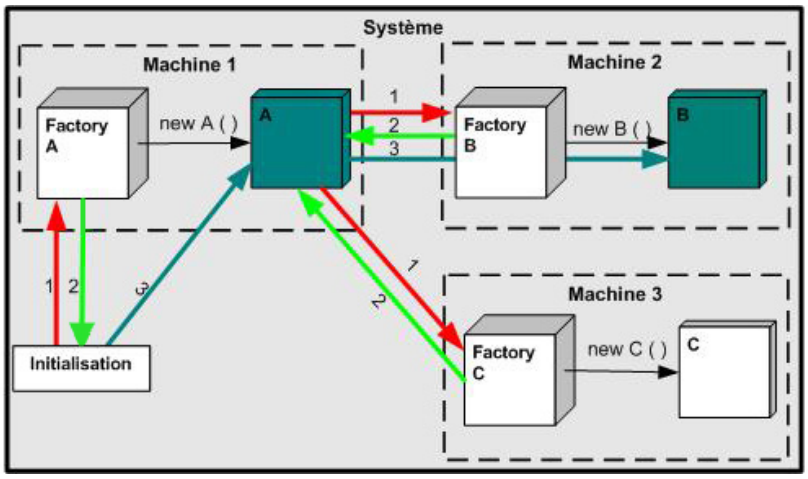

FIG. 3.13 – Déploiement et initialisation des variables.

En réalité, les composites initialisent les variables en fonction des contraintes de déploiement qui ont été imposées à la conception d'une architecture lors de l'utilisation de l'instruction *deploy*. Supposons qu'un type de composant A définit un ensemble d'instances de type B comme suit : *deploy* **2-5** B ensemble. Dès lors, il crée, via la factory de type *FactoryB*, **5** variables et en initialise uniquement **2**.

Pour les deux exemples ci-dessus, les factories *FactoryA*, *FactoryB* et *FactoryC* doivent respectivement être lancées sur les machines 1, 2 et 3. C'est la condition sine qua non qui autorise les composants composites à construire l'ensemble des variables d'un système. Ces factories sont installées et lancées par un administrateur avant d'amorcer l'initialisation de l'application comme nous venons de le montrer. Afin de déterminer l'identité des machines sur lesquelles les factories doivent être exécutées, l'administrateur utilise une matrice de déploiement.

Cette matrice est générée par un outil qui, à partir de la description d'un réseau quelconque, détermine l'emplacement de chaque variable. La figure 3.14 représente un exemple de matrice de déploiement signalant à l'administrateur les emplacements d'installation des différentes factories. Elle lui indique que des variables de type A vont être créées sur la machine 1 et 3, que des variables de type B vont être créées sur la machine 2 et enfin que des variables de type C seront déployées sur la machine 4.

	Machine 1	Machine 2	Machine 3	Machine 4
Factory A	X		X	
Factory B		X		
Factory C				X

FIG. 3.14 – Exemple de matrice de déploiement.

Comme tout objet Corba, une factory implémente des opérations définies dans une interface IDL. Le nom de cette interface correspond au nom du type de composant provenant de la description MADL. Voici un exemple d'interface IDL d'une factory :

```
interface FactoryA
{
 A createA() ;
}
```

Le naming service des factories

Les composants composites localisent les factories grâce au **naming service des factories**. Lorsqu'une factory est exécutée, elle s'enregistre dans ce naming service. Ensuite, elle est prête à recueillir les demandes de création de variables. Les conventions de nommage sont presque identiques à celles du *naming service* des connecteurs : une factory s'enregistre sous un contexte particulier suivi du nom provenant de la description MADL.

La différence réside dans la valeur de l'attribut *kind* du *component name*. En effet, cet attribut ne contient pas un numero d'identification mais bien l'identifiant de la machine, i.e l'adresse IP, où s'exécute la factory. L'attribut *kind* permet dès lors de distinguer les factories de même type enregistrées dans un même *naming context*. Deux factories de même type s'enregistrent dans le même *naming context* afin d'autoriser un type de composant d'être instantié sur des localisations physiques distinctes.

Illustrons ceci par un exemple où une factory de type *FactoryA* est responsable de la création de variables de type A sur deux machines différentes à savoir *machine 1* et *machine 2*. Deux factories de type *FactoryA* doivent donc être installées et exécutées, une sur *machine 1* et l'autre sur *machine 2*. Lorsqu'un composite cherche à créer toutes la variables de type A, il obtient la référence des deux factories en fournissant l'adresse IP de la machine sur laquelle elles se trouvent (figure 3.15). L'URL et le port d'exécution du naming service des factories sont obtenus via le résolveur.

Lorsqu'un composant déploie l'ensemble de ses sous composants, il lui est possible de sélectionner la machine sur laquelle seront hébergés ces composants en utilisant la fonction de listing fournie par le

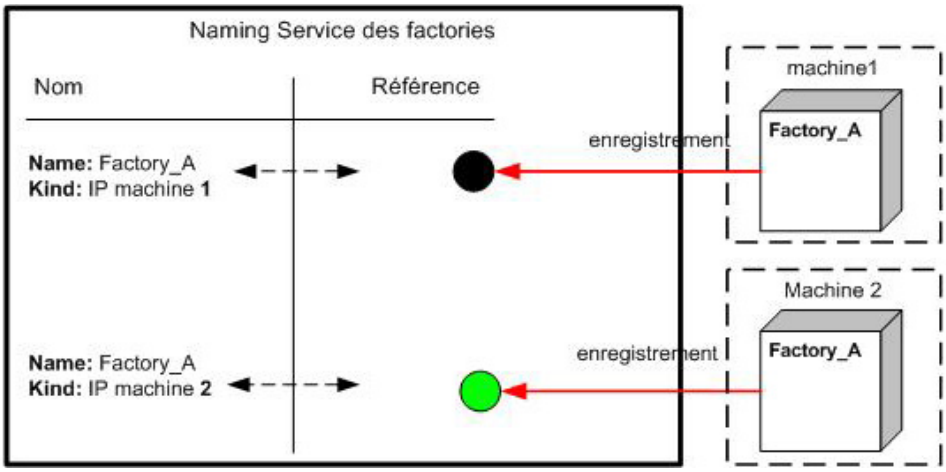

FIG. 3.15 – Enregistrement de factories dans le naming service.

naming service. En listant l'ensemble des attributs *kind* des factories enregistrés dans un *naming context* donné, le composant construit en fait une liste d'adresses IP valides pouvant héberger un nouveau composant d'un certain type.

3.3.5 Envoi et réception des requêtes

Au déploiement du système, des connexions potentielles établissent des liens entre des ensembles de variables. Lorsque ces variables sont initialisées en instances, ces connexions se matérialisent en paires de connecteurs in et out qui leur permettent d'appeler (ou recevoir) des opérations ou de déclencher (ou consommer) des événements définis dans des interfaces. Nous définissons ci-dessous le comportement adopté par un connecteur in dans le cadre d'un appel de méthode (et d'un déclenchement d'évènement) et le comportement adopté par un connecteur out pour traiter ces requêtes et renvoyer les résultats.

Appel de méthodes

Dès qu'une instance fait appel aux opérations définies dans une certaine interface unidirectionnelle, elle utilise son connecteur in pour générer une requête. Rappelons que le connecteur in a obtenu, lors de son initialisation, la référence d'un ou plusieurs connecteurs out capables d'intercepter et de comprendre cette requête. Dans le cas de connexions *manytomany* ou *manytoone*, il est tout à fait possible qu'un connecteur in possède la référence de plusieurs connecteurs out de même type. Dès lors, le connecteur in choisit la référence d'un connecteur out et l'utilise pour envoyer une requête.

La figure 3.16 illustre une connexion *manytoone* entre une instance de type *A* et trois instances de type *B*. *CI* (vert) dénote le connecteur in et *CO* (rouge) représente les connecteurs out. OI (blanc) désigne un objet d'interface implémentant les services fournis par l'interface. Supposons que chaque OI implémente l'opération suivante : **string** reverse (**string** chaîne).

Les trois connecteurs out ci-dessus sont égaux, c'est à dire qu'ils rendent identiquement les mêmes services et possèdent tous la même chance d'être sélectionnés par le connecteur in dans le cadre de l'utilisation de l'interface. Les étapes suivantes montrent le comportement de *CI* lorsque *A* désire appeler l'opération *reverse()* implémentée par les trois objets d'interfaces *OI1*, *OI2* et *OI3* :

1. L'instance *A* **appelle** la méthode *reverse()* définie par son connecteur in (*CI*).
2. *CI* **sélectionne** le premier connecteur out se trouvant dans sa table de références (*TR*). Dans notre exemple, il s'agit du connecteur out associé à *B1*, c'est à dire *CO1*.
3. *CI* **utilise** cette référence et **exécute** la méthode *reverse()* qui est aussi implémentée par *CO1*.

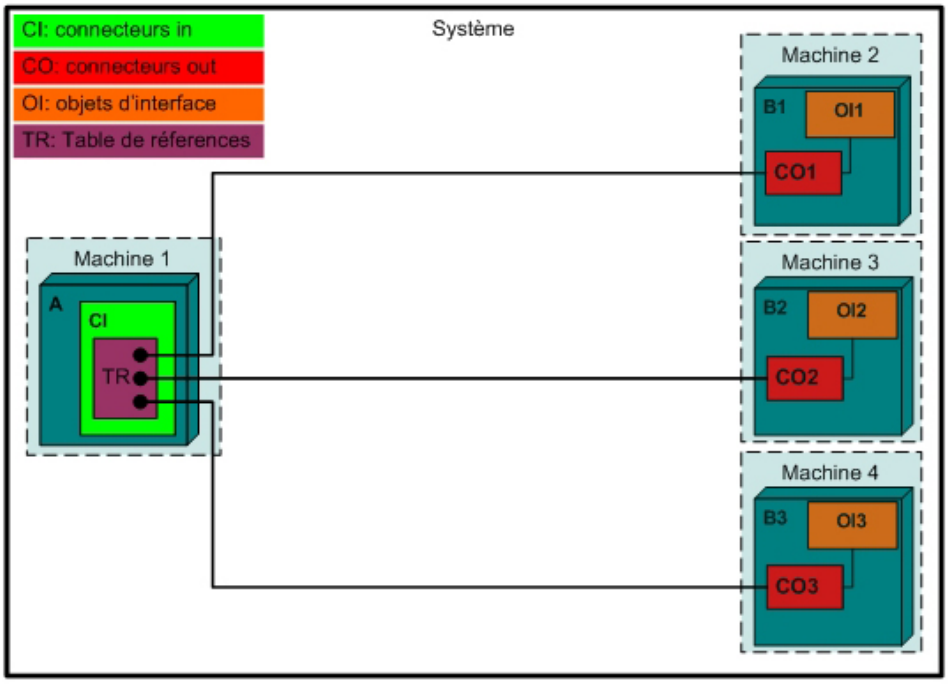

FIG. 3.16 – Connexion manytoone entre deux ensembles d'instances.

4. *CO1* reçoit la requête et vérifie si un objet d'interface est disponible. **Si aucun objet d'interface n'est disponible**, *CO1* en crée un nouveau et stocke sa référence dans la table des références des objets d'interfaces. Cependant, la création d'un nouvel objet d'interface ne peut s'effectuer que si leur nombre ne dépasse pas la cardinalité maximum définie par le contrat de type *provide*. Par exemple, l'instruction *provides 2-**5** I* ne permet pas à une instance de posséder plus de **5** objets d'interfaces de type *I*.
5. **si un objet d'interface est disponible**, CO1 exécute sur cet objet (ici OI1) la méthode *reverse()*. **Si il est impossible d'obtenir un objet d'interface**, une exception est générée et envoyée à *CI*. Dans ce dernier cas, *CI* sélectionne le connecteur out suivant, à savoir CO2. Les étapes 3 et 4 sont alors répétées.
6. Lorsque l'objet d'interface a terminé l'exécution de l'opération *reverse()*, les résultats sont envoyés à *CO1* et l'objet d'interface est désormais disponible pour accueillir d'autres requêtes.
7. *CO1* transmet ensuite les résultats à *CI*.
8. CI transmet enfin les résultats à *A*. Le processus est terminé.

Notons que les connecteurs (*CI* et *CO1*,*CO2*,*CO3*) sont susceptibles de jouer un rôle supplémentaire aux étapes 3 et 7. Par exemple, nous pouvons imaginer qu'à l'étape 3, CI encrypte les données d'entrée (le paramètre *chaîne*) et que CO2 les décrypte, à l'étape 7, afin d'appliquer correctement la méthode *reverse()* sur l'objet d'interface. Les connecteurs in et out implémentent dans ce cas des contraintes de sécurité qui leur ont été imposées lors de la description MADL, par la définition d'attributs par exemple.

De plus, nous avons supposé ici que *CI* s'est correctement initialisé et a obtenu la référence d'au minimum un connecteur out (*CO1*, *CO2* ou *CO3*). La figure 3.17 prend comme exemple deux connecteurs out, *CO1* et *CO2*, où *CO1* ne possède aucun objet d'interface de disponible. Une exception est donc envoyée à *CI*.

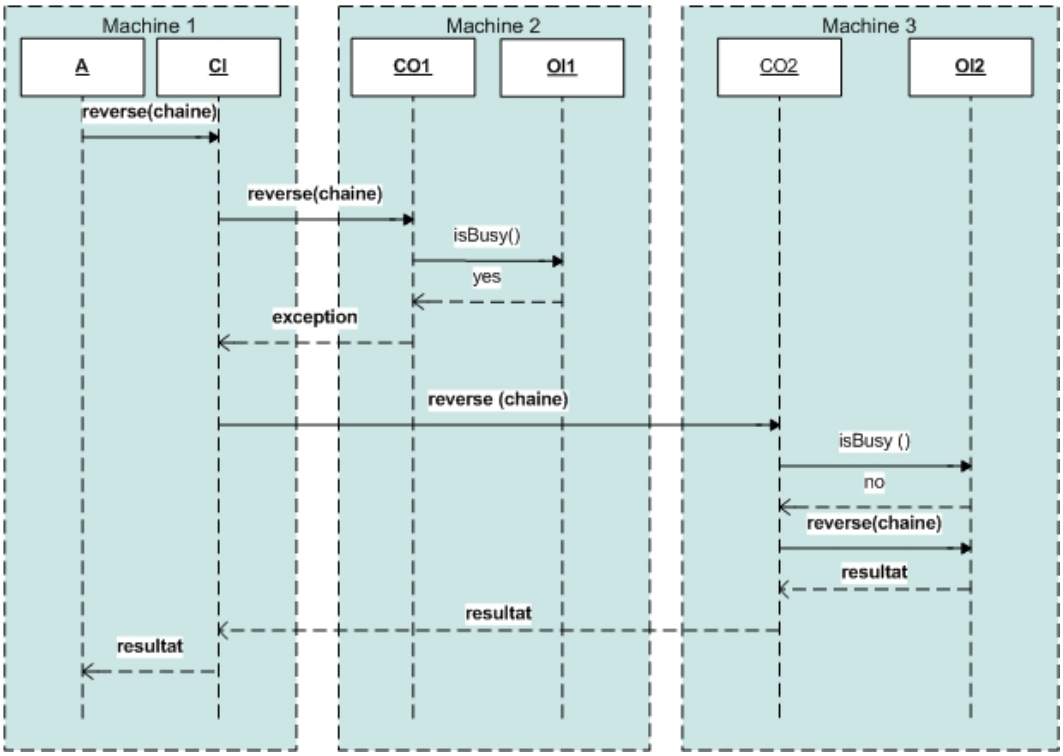

FIG. 3.17 – Appel de la méthode *reverse(chaine)*.

La méthode *reverse()* définie dans le connecteur in (flèche rentrant dans CI sur la figure 3.16) peut être implémentée en Java de la façon suivante :

```
String reverse(String chaine) {
 String reverse="" ;
 boolean envoi=false ;
 for(int i=0 ;i<3 and envoi==false ;i++) {
  try {
   reverse=TR.get(i).reverse(chaine) ;
   envoi=true ;
  }
  catch(Exception exception) {
   System.out.println("failure: request couldnt be
   sent correctly") ;
  }
 }

 if(envoi == false)
  new Exception() ;

 return reverse ;
}
```

Déclenchement d'évènements

Le déclenchement d'un évènement désigne un appel d'opérations asynchrones et non bloquantes. Quand il s'agit de communications *manytomany* ou *manytoone*, nous pouvons assimiler le déclenchement d'un tel évènement à une diffusion multiple de messages d'une instance vers d'autres.

Le processus utilisé par les connecteurs dans le cadre du déclenchement d'évènements suit une logique identique à celle exposée précédemment, à l'exception d'un point important : CI sélectionne, une à une, les références présentes dans sa table et déclenche un évènement sur chaque connecteur out (*CO1*,*CO2* et *CO3*) à partir de ces références. Changeons l'exemple précédent, désormais chaque objet d'interface implémente l'opération asynchrone suivante : **oneway** helloWorld().

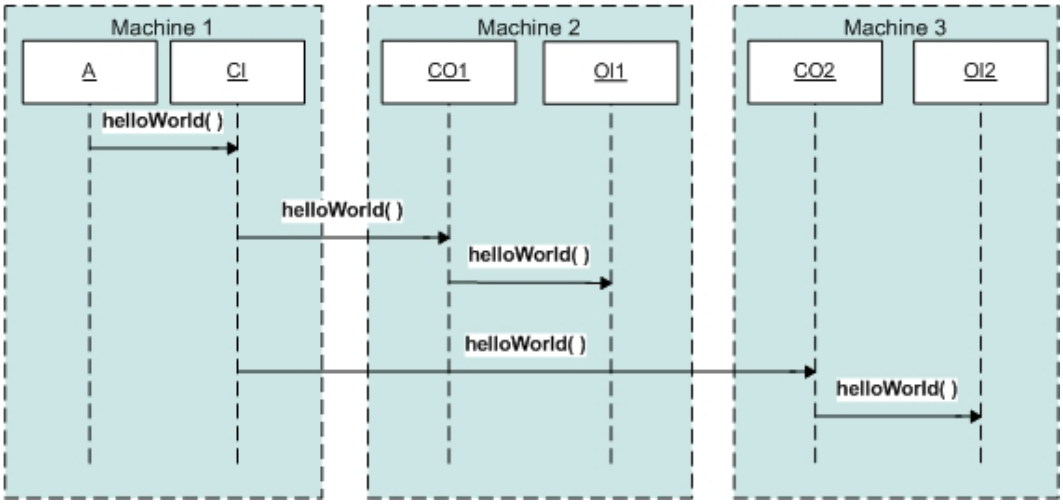

FIG. 3.18 – Déclenchement de l'évènement HelloWorld( ).

L'opération HelloWorld ( ) implémentée par le connecteur in (flèche rentrant dans CI sur la figure 1.17) peut être fournie comme ceci :

```
void HelloWorld() {
 for(int i=0 ;i<3 ;i++) {
  TR.get(i).HelloWorld() ;
 }
}
```

3.4 Un exemple : l'application client-serveur

Dans la section précédente, nous avons vu que chaque concept du langage MADL peut être transformé en structures Corba équivalentes. Nous avons aussi détaillé des pistes de solutions permettant à une application MADL de se déployer et d'établir les connexions entre les composants. Nous allons ici illustrer ces différentes règles de transformations en les appliquant à un exemple concret.

Dans un premier temps, nous présentons la définition des interfaces IDL. Nous définissons ensuite les différentes classes java générées et les relations qui existent entre elles grâce à un diagramme de classes. Finalement, nous détaillons les différentes phases importantes du cycle de vie de l'application à travers quelques séquences diagrammes.

Nous avons choisi d'illustrer le mapping vers Corba grâce à l'exemple que nous avons déjà présenté lors du chapitre précédent. Cet exemple est repris ici :

```
component_type Application <int N> {

 interface Business {
  provides operation string reverse (string chaine);
 }

 component_type Client {
  requires 1-1 Business;
 }

 component_type Server {
  provides 1-5 Business;
 }

 deploy 0-N Client as clients;
 deploy 1-1 Server as servers;
 connector onetomany link;
 connect servers.Business to link;
 for (int i=0 ; i<=N ; i++) {
  connect link to clients[i].Business;
 }
}
deploy 1-1 Application < 2 > as root.
```

3.4.1 Définition des interfaces IDL

Les types de composants Application, server et Client

Les instances du type de composant *Application* ne fournissent et requièrent aucune interface. Leur unique rôle, en tant que composite, est de créer les variables relatives aux ensembles d'instances que son type à défini, c'est à dire deux variables de type *Client* et une variable de type *Server*. Cette dernière variable doit être initialisée directement après sa création afin de refléter les contraintes de déploiement de l'ensemble d'instances nommé *servers* (*deploy **1-1** Server servers*).

L'interface IDL du type de composant *Application* définit une **opération d'initialisation** ayant pour but de créer les trois variables. Voici la définition de cette interface IDL :

```
interface Application {
 void initSubComponents();
 // crée 2 variables de type Client et une variable de type
```

```
 // Server qui est ensuite initialisée en instance.
};
```

Le type de composant *Server* fournit une interface unidirectionnelle de type *Business* connectée par le connecteur *link*. Cependant, il s'agit d'un type de composant primitif vu qu'il ne définit aucun sous composants. L'interface IDL résultante déclare dans ce cas uniquement une opération d'initialisation du connecteur out, nommée *initConnectors()*.

```
interface Server {
 void initSubComponents();
 // initConnectors () crée un connecteur out et un
 // objet d interface de type Business.
};
```

Le type de composant *Client* requière une interface unidirectionnelle de type *Business* connectée par le connecteur *link*. Cependant, il s'agit d'un type de composant primitif vu qu'il ne définit aucun sous-composants. L'interface IDL résultante déclare dans ce cas uniquement une opération d'initialisation du connecteur in, nommée *initConnectors()*. De plus, l'interface IDL *Client* définit un attribut désignant **la référence du connecteur in** permettant au futures composants de générer des requêtes.

```
interface Client {
 attribute LinkIn link;
 void initConnectors();
 // initConnectors () crée un connecteur in
};
```

Le type d'interface Business

L'interface unidirectionnelle de type *Business* fournit une opération qui accepte en entrée une chaîne de caractères, et qui retourne cette chaîne inversée. Cette interface est transformée en un objet Corba dont sa référence est détenue par les connecteurs out. L'interface IDL qui définit les services rendus par cet objet se définit comme ceci :

```
interface Business {
 string reverse(in string chaine);
};
```

La méthode *reverse()* n'est pas déclarée *oneway* car il s'agit de la définition d'une opération et non d'une action.

Les factories

Chaque type de variable est créé par une factory de type différent. L'initialisation du système nécessite l'intervention de trois types de factories standards afin de déployer les trois types de variables (*Application*, *Client* et *server*). Ces factories sont des objets Corba offrant un service de création de variables accessible à distance. Ci-dessous se trouve la définition des trois interfaces IDL décrivant les services rendus par ces objets :

```
interface FactoryApplication {
 Application createApplication();
};

interface FactoryClient {
 Client createClient() ;
};

interface Factoryserver {
```

```
 Server createServer() ;
};
```

Les connecteurs

La connexion *onetomany* est matérialisée en connecteurs in et out, nommés respectivement *linkIn* et *linkOut*. Les instances de *Client* créent un connecteur in alors que les instances de *Server* créent un connecteur out. L'interface IDL *linkIn* définit :

* une **opération d'initialisation** qui crée la connexion avec les instances de type *Server* en obtenant la référence de leur connecteur out.
* **un tableau de références de type linkOut**.
* l'opération *reverse()* qui a pour but de générer une requête.

```
Interface linkIn {
 typedef linkOut Links[];
 attribute Links links;
 String reverse(String chaine);
 void init();
};
```

L'interface IDL *linkOut* définit :

* une **opération d'initialisation** qui enregistre **la référence du connecteur out** dans le naming service.
* un ensemble de références d'objets d'interface de type *Business*.
* **l'opération reverse()** qui a pour but d'accepter les requêtes.

```
Interface linkOut {

 struct SetOfBusiness {
  Business business;
  boolean status;
 };

 typedef SetOfBusiness Interfaces[];
 attribute Interfaces interfaces;
 String requete(String chaine);
 void init();
}
```

Le fichier IDL et les modules

Les interfaces IDL présentées ci dessus sont classées dans des modules distincts afin de garder une trace des relations hiérarchiques que l'on peut retrouver dans une description d'architecture en MADL. Le principe de création des modules IDL est simple : chaque type de composant donne lieu à la création d'un module où sont rangés les interfaces IDL des interfaces unidirectionnelles et connecteurs qui y sont définis.

Voici le fichier IDL définitif issu de l'application des règles de production des interfaces IDL :

```
moduleApplication {

 interface Application {
```

```
  void initSubComponents();
 };

 interface Business {
  string reverse(in string chaine);
 };

 interface linkIn {
  typedef linkOut Connecteurs[];
  attribute Connecteurs connecteurs;
  String reverse(String chaine);
  void init();
 };

 Interface linkOut {
  struct SetOfBusiness {
   Business business;
   boolean status;
  };
  typedef SetOfBusiness Interfaces[];
  attribute Interfaces interfaces;
  String requete(String chaine);
  void init();
 };

 interface FactoryApplication {
  Application createApplication();
 };

 moduleServer {

  interface Server {
   void initSubComponents();
  };

  interface Factoryserver {
   server createserver() ;
  };

 };

 moduleClient {

  interface Client {
   attribute ConnecteurIn connecteur ;
   void initConnectors() ;
  };

  interface FactoryClient {
   Client createClient() ;
  };

 } ;

} ;
```

3.4.2 Implémentation

Cette section a pour but de présenter les différentes classes java qui vont permettre d'implémenter l'ensemble des mécanismes que nous venons de détailler. Nous pouvons distinguer deux types de classes : d'une part les classes provenant de la compilation du fichier IDL et d'autre part les classes d'implémentation. Parmi cette deuxième catégorie se trouvent les classes correspondant aux serveurs, aux implémentations des objets distribués et aux outils participant au déploiement d'une architecture MADL.

Les classes générées par le compilateur IDL

Un objet distribué possède une représentation client (*stub*) ainsi qu'une représentation serveur (*skeleton*). Une application cliente possède un *stub* pour chaque objet distribué qu'elle est susceptible d'appeler. Ce *stub* transfère les requêtes vers le serveur gérant l'objet distribué. De manière symétrique, un serveur possède un *skeleton* par objet distribué qu'il gère. Ce *skeleton* réceptionne les appels de méthode sur l'objet distribué et engendre leur exécution.

La compilation du fichier IDL (présenté à la fin de la section précédente) vise à générer l'ensemble des *stubs* et *skeletons* correspondant aux types de composants, interfaces unidirectionnelles et connecteurs. Les *stubs* sont créés par une classe particulière. Ces classes Java vont permettre aux connecteurs d'envoyer et recevoir des requêtes sur un bus ORB.

Un package est généré pour chaque module du fichier IDL et contient les *stubs* et *skeletons* des objets distribués définis.

Les serveurs de gestion des objets distribués

Pour rappel, la norme CORBA se base sur le modèle *client/serveur*. Les objets que l'on souhaite rendre accessible à distance sont crées et gérés par un serveur CORBA. Son rôle est d'exécuter les méthodes appelées par les applications clientes sur les objets distribués.

Par exemple, la classe *StartServerApplication.java* crée une instance de type *Application* et initialise le bus ORB afin de permettre à cet instance de recevoir des appels de méthode.

Les implémentations d'objets distribués

Une implémentation est fournie pour chaque opération définie dans l'interface d'un objet CORBA. Ces opérations sont implémentées au sein d'une classe java particulière. Par exemple, la classe suivante vise à implémenter l'objet distribué désignant le type d'interface *Business* :

```
public class BusinessImpl extends BusinessPOA {
// BusinessPOA désigne le skeleton associé à
// l objet distribué de type Business

 public string reverse(string chaine) {
  return new StringBuffer(chaine).reverse().toString() ;
 }

}
```

Nous verrons dans le chapitre suivant que l'implémentation des services définis par les connecteurs et les types de composants peuvent être générés automatiquement sur base des informations contenues dans la description d'une architecture MADL. L'implémentation des objets d'interfaces doit être fourni par le programmeur.

Les classes "*outils*"

Les classes *outils* fournissent des services d'aide au déploiement d'un système. Parmi ces classes, nous retrouvons :

* **Le résolveur**. Cette classe est un serveur UDP connaissant l'url du naming service.
* **Le constructeur de naming service**. Cette classe initialise le naming service en créant les *naming contexts* adéquats selon la composition hiérarchique de l'architecture déployée.
* **Le déployeur de composants**. Cette classe déploie un composant d'un certain type. Il liste les factories adéquates disponibles dans le *naming service*, et propose à l'utilisateur un choix de localisation du futur composant.
* **L'explorateur de contextes**. Cette classe fournit une méthode d'exploration de *naming contexts*. Il renvoie, sous forme de vecteur, tous les *name components* enregistrés dans un *naming context* sur base d'un chemin d'accès donné.

3.4.3 Diagramme de classes

Le diagramme de classes suivant présente un aperçu des classes java qui composent l'application déployée sur trois machines différentes. Ce diagramme utilise la notation UML pour définir les relations entre les différentes classes. Chaque couleur du diagramme désigne la machine sur laquelle les classes sont exécutées. Dans notre exemple, les objets sont distribués sur trois machines : *machine 1*, *machine 2* et *machine 3* (figure 3.19).

3.4.4 Diagrammes de séquences

Les diagrammes suivants illustrent le comportement des objets distribués (les types de composants, les connecteurs et objets d'interfaces) pour les cas d'utilisations suivants :

* **Installation des factories**

* **Création des variables**

* **Initialisation d'une variable**

* **Utilisation de l'interface Business**

Nous supposons ici que les objets ont obtenu la référence du *naming service* grâce au résolveur.

L'installation des factories

L'administrateur lance les différents types de factories sur les machines du réseau en respectant la **matrice de déploiement** (figure 3.20).

Les factories, une fois exécutées sur l'ensemble des trois machines, sont prêtes à créer différents types de variables. Nous pouvons observer que le *naming service* des factories est hébergé par la *machine 3*.

La création des variables

La classe d'initialisation du système peut être lancée sur une machine au choix. Via le *naming service* des factories, cette classe obtient la référence de la factory lancée sur la machine 1 (celle qui crée des variables de type *Application*). Cette référence est ensuite utilisée pour créer une première variable en exécutant l'opération *createApplication()*. Comme cette variable doit obligatoirement être initialisée (deploy **1**-1 Application), les opérations *initSubComponents()* et *initConnectors()* sont ensuite appelées. Le processus est répété de manière récursive comme nous l'avons détaillé précédemment dans la section

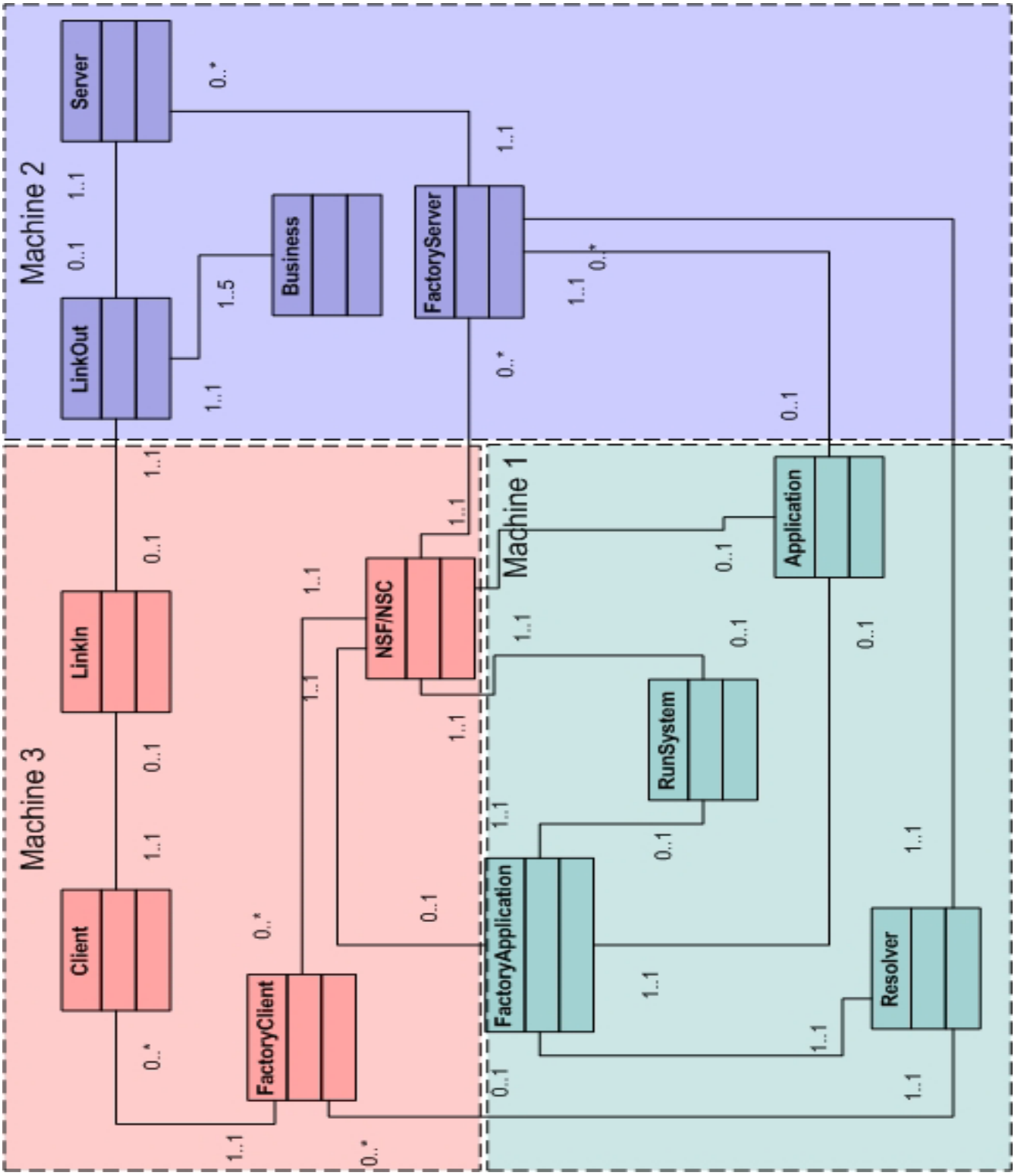

FIG. 3.19 – Diagramme de classes.

	Machine 1	Machine 2	Machine 3
FactoryApplication	X		
FactoryServer		X	
FactoryClient			X

FIG. 3.20 – Matrice de déploiement des factories.

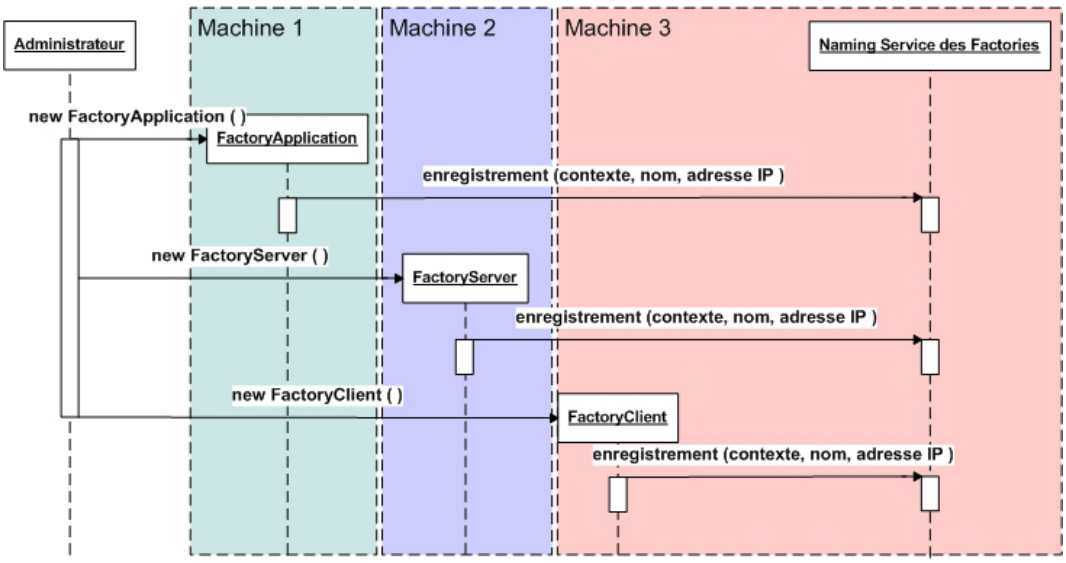

FIG. 3.21 – Lancement des Factories.

3.3.4.

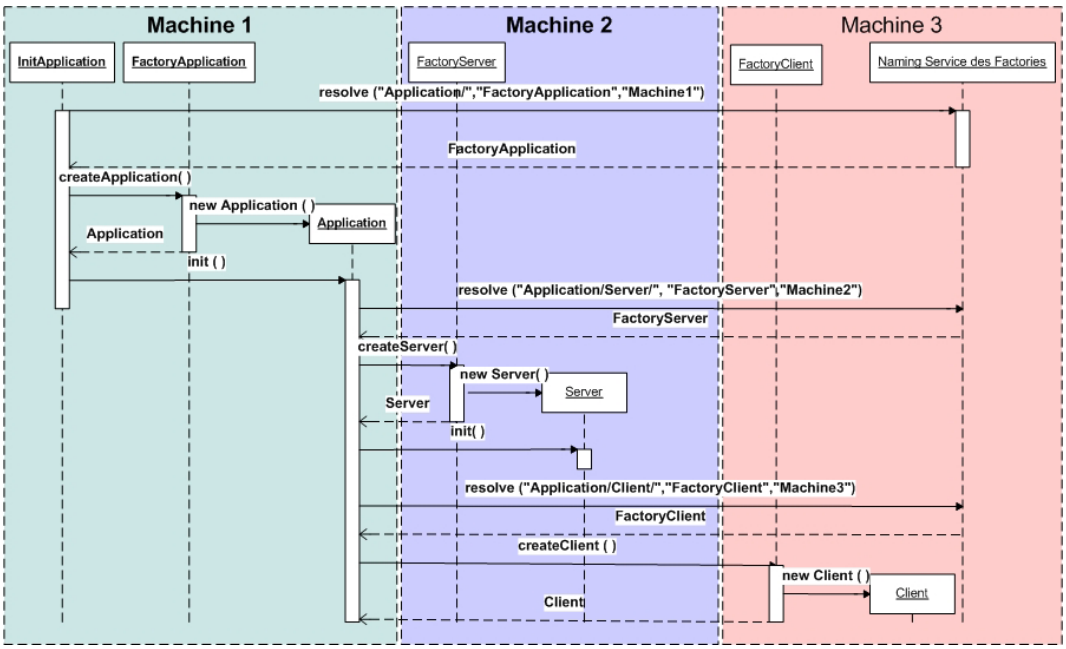

FIG. 3.22 – Création des variables du système par récursivité.

L'initialisation des instances

Les deux diagrammes suivants détaillent l'initialisation d'une variable de type *Server* et *Client*. La connexion entre les clients et le serveur a été définie dans le contexte nommé "Application/".

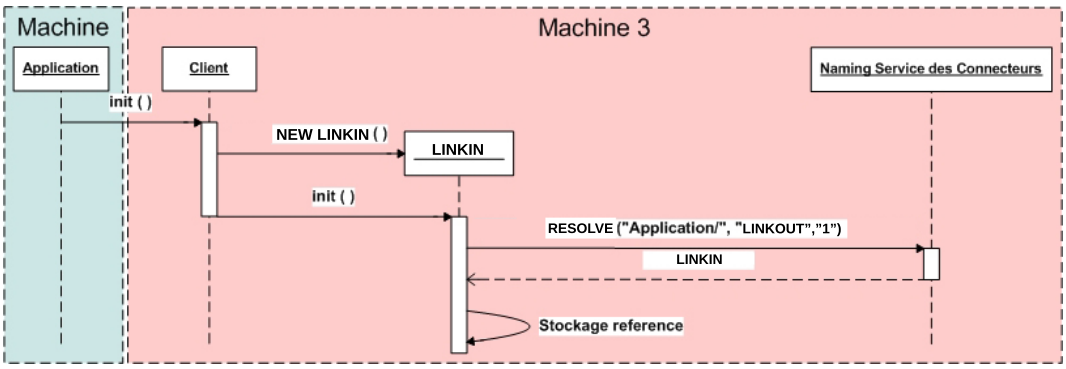

FIG. 3.23 – Initialisation d'une variable de type *Client*.

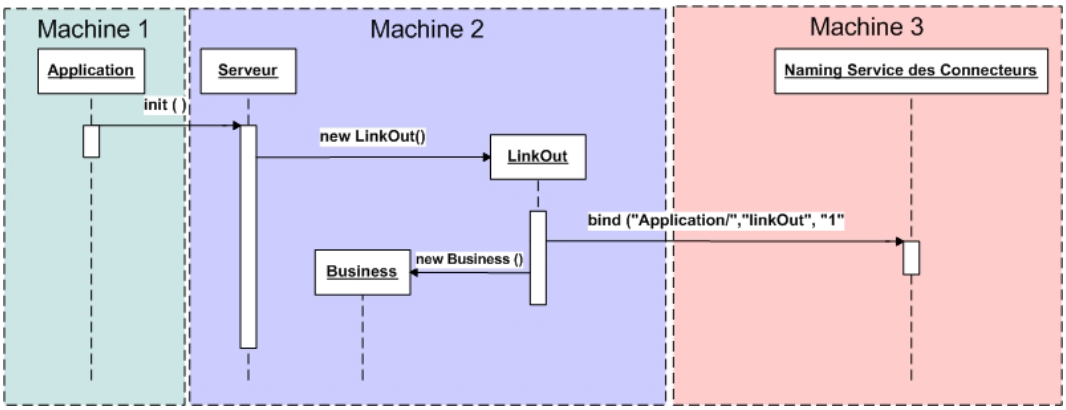

FIG. 3.24 – Initialisation d'une variable de type *Server*.

L'utilisation de l'interface Business

Ce diagramme détaille l'acheminement d'une requête visant à utiliser l'opération *reverse()* définie par l'interface *Business* (figure 3.25).

3.5 Approfondissement

3.5.1 Les interfaces bidirectionnelles

Jusqu'à présent, nous avions émis l'hypothèse que les interfaces étaient toutes unidirectionnelles. Cependant, MADL propose une version plus originale de la notion d'interface. En effet, une interface en MADL peut être bidirectionnelle, c'est à dire qu'elle peut aussi bien fournir des opérations ou actions qu'en requérir. Ainsi le composant qui fournit cette interface peut aussi initier des communications vers d'autres composants en utilisant le même protocole, la même interface.

Corba IDL définit uniquement la notion d'interfaces unidirectionnelles comportant la description d'opérations synchrones ou asynchrones qui peuvent être fournies par un objet. La première étape de transformation des interfaces bidirectionnelles revient donc à proposer un équivalent en termes d'interfaces IDL. L'idée est de créer deux interfaces IDL différentes où la première interface contient les opérations et actions requises et où la seconde contient les opérations et actions fournies par l'interface bidirectionnelle.

Chacune de ces deux interfaces IDL définit donc une partie des opérations et actions définies dans l'interface initiale. Le type de composant qui requiert l'interface initiale va désormais fournir une partie

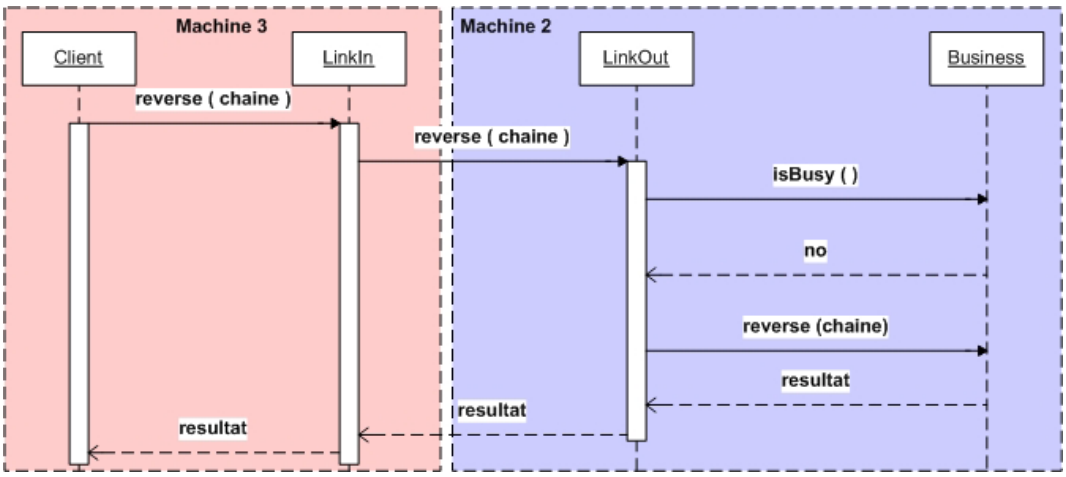

FIG. 3.25 – Utilisation de l'opération *reverse()* définie dans l'interface *Business*.

de celle ci, c'est à dire la première interface IDL. Par contre, le type de composant qui fournit l'interface de départ, va fournir la seconde interface IDL. L'interface bidirectionnelle a donc été divisée en deux interfaces IDL unidirectionnelles. Si nous suivons notre raisonnement en termes de connecteurs, cela signifie que chaque composant va désormais disposer d'un connecteur in et d'un connecteur out dans le cadre de l'utilisation de la même interface.

Afin de garder une sémantique identique à celle associée aux interfaces bidirectionnelles, ces deux interfaces IDL doivent continuer à former un protocole commun. Ce protocole commun va autoriser n'importe quel composant impliqué dans celui ci d'initier une communication sur des bases identiques. Pour ce faire, le connecteur in et out associé à un composant doivent former un tout.

Ainsi, chaque composant dispose maintenant d'un **connecteur entier**, rassemblant les deux rôles in et out. Le rôle in lui permet d'utiliser la partie de l'interface bidirectionnelle fournie par d'autres composants, alors que le rôle out lui permet de recevoir des requêtes en provenance d'autres composants qui désirent utiliser l'autre partie de l'interface bidirectionnelle.

Illustrons notre raisonnement par un petit exemple. Voici la définition d'une interface bidirectionnelle et de deux types de composants. L'un requière cette interface et l'autre la fournit.

```
interface I {
 provides operation void A () ;
 requires operation void B () ;
}

type_component Client {
 requires 1-1 I ;
}

type_component server {
 provides 1-1 I ;
}
```

La première étape consiste à transformer l'interface *I* en deux interfaces IDL *I1* et *I2*. *I1* contient la définition des opérations requises par *I* et *I2* contient la définition des opérations fournies par *I*.

```
interface I1 {
 void A ();
}

interface I2 {
 void B ();
}
```

Pour garder l'équivalence, le composant client doit requérir *I1* et fournir *I2* alors que le composant Seveur doit requérir *I2* mais fournir *I1*. Si une connexion existe entre deux instances de ces types de composants, ils devront chacun d'eux disposer de connecteurs in et out. En effet, l'instance de *Client* disposera d'un connecteur out pour recevoir des requêtes de l'instance de *Server* qui désire utiliser l'opération *B* définie dans *I2* et d'un connecteur in pour utiliser la méthode *A* définie dans *I1*. Ceci correspond bien à la sémantique de départ qui spécifie que le client peut utiliser la méthode A et doit implémenter le méthode *B*. Il implémente effectivement la méthode *B* vu qu'il fournit désormais l'interface *I2*.

Les connecteurs in et out associés à chacune des deux instances opèrent désormais dans le cadre d'un même connexion et de l'utilisation d'une même interface de départ. C'est pourquoi ces deux types de connecteurs sont fusionnés pour n'en former qu'un seul. Désormais, le client ou le serveur peuvent initier une communication sur des bases égales, ce qui correspond à la signification que nous avons associée à l'interface bidirectionnelle.

En termes d'interfaces IDL, la modification majeure s'opère au niveau des connecteurs. La distinction entre les connecteurs in et out disparaît, et désormais chaque composant dispose de connecteurs définissant les deux rôles. Ces connecteurs s'enregistrent dans le naming service de la même manière et initialisent les références des connecteurs homologues aussi de façon identique au procédé que nous avons défini.

Modifions un peu l'exemple du *client-serveur* de la section précédente. L'interface *Business* est désormais bidirectionnelle et se définit comme suit :

```
interface Business {
 provides operation string reverse (string chaine);
 requires operation string concat (string chaine1,string chaine2);
}
```

Si nous appliquons ces nouvelles règles, nous obtenons désormais les interfaces IDL ci-dessous. Nous obtenons premièrement deux interfaces IDL définissant d'une part l'opération *reverse()* et d'autre part l'opération *concat()*, que l'on nomme respectivement *BusinessClient* et *BusinessServer*.

```
interface BusinessServer {
 string reverse(string chaine);
}

interface BusinessClient {
 string concat(string chaine1, string chaine 2);
}
```

Les connecteurs englobent désormais les deux rôles in et out. Nous les avons nommés *linkClient* et *linkServer*. *linkClient* dénote le connecteur associé au client alors que *linkServer* représente le connecteur associé au serveur. La différence entre les deux connecteurs réside dans le type de l'interface qu'ils fournissent. *linkClient* détient des références vers des objets de type *BusinessClient* alors que *linkServer* détient des références vers des objets de type *BusinessServer*.

```
Interface linkClient {

 struct SetOfBusiness {
  BusinessClient business;
  boolean status;
 };

 typedef SetOfBusiness Business[] ;
 attribute Business business;
 typedef linkServer Connecteurs[];
 attribute Connecteurs connecteurs[] ;
 String requete(string chaine) ;
 String concat(string chaine1,string chaine2) ;
 void init() ;
}

Interface linkServer
{

 struct SetOfBusiness {
  BusinessServer business ;
  boolean status ;
 }

 typedef SetOfBusiness Business[] ;
 attribute Business business;
 typedef linkClient Connecteurs[];
 attribute Connecteurs connecteurs[] ;
 String concat(string chaine1,string chaine2) ;
 String requete(string chaine) ;
 void init() ;
}
```

3.5.2 Les connexions de délégation

Un connecteur de délégation ne crée aucun objet d'interface à son initialisation. Cependant, lorsqu'il intercepte une requête, il obtient la référence du connecteur du sous-composant vers lequel la connexion est déléguée. Cette référence est ensuite utilisée pour déléguer l'appel de méthode.

En termes d'interface IDL, un connecteur de délégation entraîne la déclaration d'un attribut supplémentaire : la référence du connecteur du sous-composant.

Chapitre 4

Réalisation d'un prototype : Validation des règles de mapping

L'intérêt de ce quatrième chapitre est de valider la faisabilité des règles de mapping que nous avons dégagées en développant un prototype d'implémentation. Ce prototype vise à valider les mécanismes permettant le déploiement d'une architecture décrite en MADL, à savoir :

* **Le métamodèle**. Nous allons observer si le métamodèle, ainsi défini, nous permet d'implémenter les règles de mapping.

* **La génération du fichier IDL**. Le prototype va montrer la faisabilité des règles de production des interfaces IDL relatives aux données présentes dans le métamodèle.

* **L'initialisation du naming service**.

* **La localisation des composants**. Nous allons montrer que le mécanisme détaillé permettant aux composants de se localiser est tout à fait réalisable et peut être implémenté par notre prototype.

* **Le déploiement de l'architecture**. Le déploiement récursif des composants est une séquence d'appels sur des objets distribués dont la référence est obtenue par l'intermédiaire de factories. Ce comportement implémenté par chaque composant est implémenté par le prototype et permet de valider notre approche.

* **La communication entre les composants**. Finalement, nous démontrons que le système, une fois déployé, permet aux composants de communiquer ensemble sur le bus de communication CORBA.

On ne considère pas ici les interfaces bidirectionnelles, ni les connecteurs de délégation. Vu que la gestion des types ne représente pas un point important de ce mémoire, uniquement les paramètres de type *String* seront utilisés. Les attributs d'architecture ne font également pas partie du domaine étudié. Finalement, nous avons choisi le langage java comme langage de programmation et *openorb* comme distribution CORBA.

4.1 Principes de conception

Le développement de ce prototype est un processus qui se déroule en 3 phases :

1. Création et instanciation du référentiel.
2. Génération et compilation du fichier IDL.

3. Génération du code des classes Java.

Sur la figure 4.1, nous pouvons constater que le référentiel est utilisé comme source de données aux étapes 2 et 3. Lorsque l'étape 2 est terminée, des packages sont créés pour contenir les *stubs* et les *skeletons* des interfaces IDL générées. L'étape 3 génère les classes qui vont définir le comportement du système et les enregistre dans les packages adéquats. Lorsque la dernière phase est exécutée, nous pouvons déployer un système conforme à l'architecture décrite en MADL à l'aide de tous les fichiers générés.

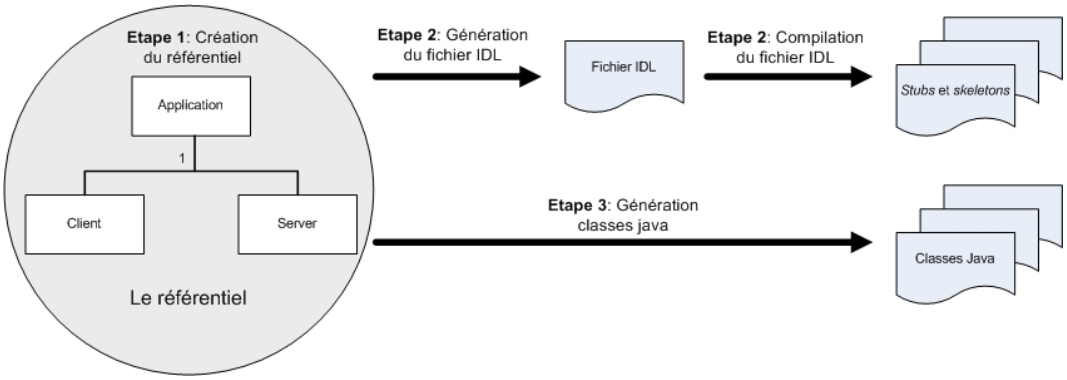

FIG. 4.1 – Etapes du développement du prototype.

4.2 Le référentiel

Le référentiel est un ensemble de classes Java qui proviennent du métamodèle et qui sont destinées à contenir l'information liée à la description d'une architecture en MADL. Ce référentiel constitue la source de données à partir de laquelle le prototype va pouvoir exécuter la génération de code.

Pour simplifier le problème, ce référentiel est instantié à la main sur base du simple exemple du *client-serveur*. Le métamodèle a aussi été simplifié pour s'adapter au limites du domaine étudié dans notre approche. Ainsi les classes représentant les attributs d'architecture et les connexions de délégation ont été supprimées du métamodèle. De plus, les opérations et actions font désormais partie uniquement d'interfaces unidirectionnelles.

Les relations *0-N* ou *1-N* sont représentées par des attributs de type *Vector*. L'exemple ci-dessous montre l'instanciation d'une partie du référentiel, représenté par la classe *Metamodel.java*. Les classes du référentiel sont enregistrées dans un package spécifique nommé *metamodel*.

```
// création des types de composants
application=new ComponentType("Application",null) ;
server=new ComponentType("Server",application) ;
client=new ComponentType("Client",application) ;
application.getSubcomponents().add(client) ;
application.getSubcomponents().add(server) ;

//creation de l interface Business
business = new InterfaceType("Business") ;
parameter = new Parameter("chaine","in","string") ;
parameterResult = new Parameter(null,"result","string") ;
reverse = new Operation("reverse") ;
reverse.getParameters().add(parameterResult) ;
reverse.getParameters().add(parameter) ;
business.getFeatures().addElement(reverse) ;
application.getInterfaces().add(business) ;
```

```
// creation des contrats
provide = new Provide("busser",0,3,business) ;
require = new Require("buscli",1,1,business) ;
server.getContracts().add(provide) ;
client.getContracts().add(require) ;

// création des connecteurs
link = new One_to_many ("link") ;
application.getConnectors().add(link) ;
```

Ce référentiel peut être perçu comme un arbre où chaque noeud désigne un type de composant. Un noeud contient des références vers les classes correspondant aux interfaces, connecteurs ou ensembles d'instances qui ont été définis à l'intérieur du type de composant. L'attribut *rootComponent* défini dans la classe *Metamodel.java* désigne la racine de cet arbre, i.e le type de composant racine de l'architecture.

La méthode suivante permet le parcours de l'arbre de manière récursive :

```
voidvisitNode(ComponentType node) {

 // traitement du noeud
 trtNode(node);

 if(node.getSubcomponents() != null) {
  for(int i=0 ;i < node.getSubcomponents().size() ;i++) {
   visitNode((ComponentType)node.getSubcomponents().get(i)) ;
  }

 }

}
```

De cette manière, nous allons parcourir toute la description de l'architecture afin de générer l'ensemble des interfaces IDL dans un premier temps et le code des autres classes Java dans un second temps.

4.3 Génération et compilation du fichier IDL

Le prototype parcoure l'ensemble du référentiel et génère, pour chaque noeud (représenté par l'attribut *node*), les interfaces IDL représentant :

1. Au **type de composant** désigné par le noeud.
2. Aux **types d'interfaces unidirectionnelles** définies par le type de composant.
3. Aux **connecteurs** définis par le type de composant.
4. A la **factory** responsable de créer des composants dont le type est désigné par le noeud.

4.3.1 Les interfaces unidirectionnelles

Ces interfaces IDL portent le nom donné à l'interface unidirectionnelle lors de la description de l'architecture. *fileIDL* est un attribut de type *FileCreator*. Cette classe gère la création, l'écriture et finalement l'enregistrement d'un fichier dans un package adéquat. Voici le code permettant la génération de ces interfaces :

```
if(node.getInterfaces() != null) {

 for(int i=0 ;i < node.getInterfaces().size() ;i++) {
  InterfaceType interfaceT = (InterfaceType)node.getInterfaces()
                             .get(i) ;
  fileIDL.write("interface " + interfaceT.getName() + " {") ;

  //génération des services définis par l'interface
  if(interfaceT.getFeatures() != null)
   printFeatures(interfaceT,fileIDL) ;

  fileIDL.write("} ;") ;
 }

}
```

La méthode *printFeatures()* génère le code relatif aux services et actions définis dans l'interface MADL.

4.3.2 Le type de composant

Chaque noeud de l'arbre désigne un type de composant composite ou primitif. La difficulté concernant la génération de ces interfaces IDL réside dans l'obtention du nom des connecteurs qui connectent les interfaces requises par le type de composant. Ces noms sont utilisés pour définir les attributs représentant les connecteurs in d'un type de composant. Le code ci-dessous montre que cela est obtenu en parcourant l'ensemble des ports associés au contrats de type *Require* définis par le type de composant requièrant une interface. Depuis ces ports, il est aisé d'obtenir le nom du connecteur. La méthode *foundElement()* évite l'ajout de plusieurs connecteurs de même nom. En effet, un contrat peut être associé à plusieurs ports dont le connecteur est identique.

```
fileIDL.write("interface " + node.getName() + " {") ;

// génération des attributs désigant la référence
// des connecteurs in si le type de composant
// requièrt des interfaces connectées
Vector connectorsIn = new Vector() ;
if(node.getContracts() != null) {

 for(int i=0 ;i<node.getContracts().size() ;i++) {
  Contract contract = (Contract)node.getContracts().get(i) ;

  if(contract.getClass().getSimpleName().compareTo("Require") == 0)
  {

   if(contract.getPorts() !=null) {

    for(int j=0 ;j<contract.getPorts().size() ;j++) {
     Port port = (Port)contract.getPorts().get(j) ;

     if(foundElement(connectorsIn,port.getConnector().getName())
     == false)
     {
      connectorsIn.add(port.getConnector().getName()) ;
     }
    }
   }
```

```
  }
 }
}
for(int i=0 ;i<connectorsIn.size() ;i++) {
 fileIDL.write(" attribute " + (String)connectorsIn.get(i) + "In "
          + ((String)connectorsIn.get(i)).toLowerCase() + " ;") ;
}
// génération des méthodes d initialisation
// du type de composant
fileIDL.write(" void initSubComponents() ;") ;
fileIDL.write(" void initConnectors() ;") ;
fileIDL.write("} ;") ;
```

4.3.3 Les connecteurs

Chaque déclaration de connecteur à l'intérieur d'un type de composant donne lieu à la définition de deux interfaces IDL qui dénotent respectivement son rôle in et out. Dans les deux cas, le prototype a besoin d'accéder aux contrats pour connaître le nom de l'interface faisant l'objet de la connexion. Ce sont les premiers ports (indice numéro 0), issus des relations *from* et *to* du référentiel, qui sont utilisés pour accéder aux contrats *Require* et *Provide*. L'attribut *con* désigne la référence du connecteur courant.

La rôle out de chaque connecteur est généré de la manière suivante :

```
String interfaceName = ((Contract)((Port)con.getTo().get(0))
             .getContract()).getInterfaceType().getName();
fileIDL.write("interface " + con.getName() + "Out {");
fileIDL.write(" struct SetOf" + interfaceName + " {");
fileIDL.write(" " + interfaceName + " IO" + " ;");
fileIDL.write(" boolean status ;");
fileIDL.write(" } ;");
fileIDL.write(" typedef SetOf" + interfaceName + " Interfaces["
        + ((Contract)((Port)con.getTo().get(0)).getContract())
                          .getCardinality_max() + "] ;");
fileIDL.write(" attribute Interfaces interfaces ;");

// Génération des signatures des services définis
// dans l interface connectée
printFeatures(((Contract)((Port)con.getFrom().get(0))
       .getContract()).getInterfaceType(),fileIDL);
fileIDL.write(" void init() ;");
fileIDL.write("} ;");
```

La rôle in de chaque connecteur est généré de la manière suivante :

```
fileIDL.write("interface " + con.getName() + "In {");
fileIDL.write(" typedef " + con.getName() + "Out ConnectorOut["+
((Contract)((Port)con.getFrom().get(0)).getContract())
.getCardinality_max() + "] ;");
fileIDL.write(" attribute ConnectorOut connectorOut ;");

// Génération des signatures des services définis
// dans l interface connectée
printFeatures(((Contract)((Port)con.getFrom().get(0))
       .getContract()).getInterfaceType(),fileIDL);
fileIDL.write(" void init() ;");
fileIDL.write("} ;");
```

4.3.4 La factory

Finalement, pour chaque *node*, la factory correspondante est générée de la manière suivante :

```
fileIDL.write(" interface Factory" + node.getName() + "{") ;
fileIDL.write(node.getName() + " create"+node.getName()+"( );") ;
fileIDL.write(" } ;") ;
```

4.3.5 La compilation du fichier idl

Le prototype utilise la classe *ProcessLauncher* afin d'exécuter automatiquement la compilation du fichier IDL. Cette classe permet de lancer une application externe en consommant les divers fluxs dans des threads séparés. *ProcessLauncher* utilise la méthode *exec* fournie par la classe *Runtime*.

```
ProcessLauncher process = new ProcessLauncher() ;
try {
 System.out.println("Compilation du fichier IDL...") ;
 process.exec(new String[]{"cmd.exe", "/c","D :/Fundp/Prototype
                          /Compilateur/tools/compileIDL.bat"}) ;
 System.out.println("Fichier IDL compilé...") ;
}
catch (IOException ex) {
 ex.printStackTrace() ;
}
```

Ce bout de code appelle donc automatiquement le fichier batch *compileIDL.bat* qui fait appel à l'outil *idl2java* :

```
cd d :/Fundp/Prototype/Compilateur/tools/
idl2java ../generator/architecture.idl -d ../
```

Après la compilation, les packages relatifs aux modules IDL sont créés et contiennent les *stubs* et *skeletons* des interfaces IDL. La troisième phase du processus peut dès lors commencer.

4.4 La génération de code

Le développement du prototype se poursuit par la dernière phase qui consiste à générer le code des différentes classes Java qui vont doter les différentes structures de l'architecture d'un comportement lui permettant de prendre vie. Ces classes implémentent d'une part, les services rendus par les objets distribués (types de composants, interfaces et connecteurs) et les *outils* (initialisation et exploration du naming service, déploiement des composants, résolution).

4.4.1 Les objets distribués

Le référentiel est parcouru une seconde fois dans son entièreté à partir du noeud racine. Une classe d'implémentation et une classe *serveur* sont générées pour :

* Le **type de composant** désigné par le noeud courant.

* Les **interfaces unidirectionnelles** définies dans le noeud courant.

* Les **connecteurs** in et out définis dans le noeud courant.

* La **factory** responsable de l'instanciation du type de composant désigné par le noeud.

Les classes *Serveurs*

Une classe *Serveur* active un objet distribué au près du POA. Il rend ensuite accessible la référence de l'objet, sous forme d'IOR, pour permettre à d'autres applications de la résoudre en *stub*.

Les classes *serveurs* sont à peu près toutes identiques, à la différence qu'ils activent et gèrent un type d'objet distribué distinct. Voici un exemple de classe générée par le prototype qui permet l'activation d'un composant de type *Application* :

```
public class StartApplication extends Thread {
 org.omg.CORBA.ORB orb ;
 String IOR ;
 public StartApplication() {
  orb = null ;
  IOR = "" ;
  try {
   orb = org.omg.CORBA.ORB.init(null,null) ;
   ApplicationImpl component = new ApplicationImpl() ;
   org.omg.CORBA.Object objPoa = orb.resolve_initial_references
                                  ("RootPOA");
   org.omg.PortableServer.POA rootPOA rootPOA = org.omg.Portable
                                 Server.POAHelper.narrow(objPoa);
   byte[] servantName = rootPOA.activate_object(component) ;
   org.omg.CORBA.Object objPl = rootPOA.id_to_reference
                                 (servantName) ;
   IOR = orb.object_to_string(objPl) ;
   rootPOA.the_POAManager().activate() ;
  {
  catch ( org.omg.CORBA.SystemException e ) {
   System.out.println("Exception : exception CORBA") ;
   e.printStackTrace() ;
  }
  catch ( Exception e ) {
   System.out.println("Exception : undefined code") ;
   e.printStackTrace() ;}
  }

 synchronized public void run() {
  orb.run() ;}

 public org.omg.CORBA.ORB getOrb() {
  return orb ;}

 public String getIOR() {
  return IOR ;}
}
```

Ce type de classe est exécuté au sein d'un thread séparé afin d'éviter le blocage des applications appelantes. Lorsqu'une application, comme les factory par exemple, lance ce type de classe, elle définit le code suivant :

```
Application servant=null ;
StartApplication startApplication = new StartApplication() ;
org.omg.CORBA.Object obj = startApplication.getOrb()
    .string_to_object (startApplication.getIOR());
startApplication.start() ;
servant = ApplicationHelper.narrow(obj) ;
```

Nous pouvons remarquer qu'un serveur retourne l'IOR de l'objet distribué activé vers l'application

appelante. Cette dernière l'utilise afin de créer un *stub* permettant d'envoyer des requêtes vers cet objet distribué. Ainsi, les factories, peuvent retourner la référence d'un *stub* vers un type de composant sans que ce dernier nécessite l'emploi du *naming service*.

Les classes d'implémentation

Une classe d'implémentation fournit le code qui va définir le comportement des composants, des interfaces qu'ils fournissent et de leurs connecteurs. Ce type de classe étend le *skeleton* généré par le compilateur IDL (extension POA) et implémente les services déclarés dans les différentes interfaces IDL.

La classe suivante implémente un composant de type *Server*. Le prototype génère automatiquement une implémentation pour les méthodes d'initialisation *initSubComponents()* et *initConnectors()*. Nous pouvons remarquer que l'initialisation des connecteurs out utilise le mécanisme que nous venons de présenter : l'IOR du connecteur out est retournée à l'implémentation du composant de type *Server* qui peut ensuite la résoudre pour créer un *stub* lui permettant d'appeler la méthode *init()* sur ce connecteur out, désormais activé. Ce processus correspond à ce que nous avons expliqué dans la section 3.3.3.

```
public class ServerImpl extends ServerPOA {

 public void initSubComponents() {
 // No implementation because this component
 // type doesnt define any subcomponents
 }

 public void initConnectors() {
  StartlinkOut startlinkOut = new StartlinkOut() ;
  org.omg.CORBA.Object obj = startlinkOut.getOrb()
       .string_to_object (startlinkOut.getIOR()) ;
  startlinkOut.start() ;
  linkOut linkOut = linkOutHelper.narrow(obj) ;
  linkOut.init() ;
 }

}
```

4.4.2 Les classes *outils*

Le resolveur

Cette classe n'est pas générée depuis le référentiel mais constitue un outil autorisant les composants et les connecteurs déployés à résoudre l'adresse du *naming service* de manière transparente.

```
public class Resolver {
 String NSAddress ;
 public Resolver(String NSAddress) {
  this.NSAddress = NSAddress ;
 }

 public void runServerUDP() throws Exception {
  MulticastSocket serverSocket = new MulticastSocket(6000) ;
  serverSocket.joinGroup(InetAddress.getByName("231.1.2.3")) ;
  byte[] receiveData = new byte[20] ;
  byte[] sendData = new byte[NSAddress.getBytes().length] ;
  while (true) {
   DatagramPacket receivePacket = new DatagramPacket
   (receiveData, receiveData.length) ;
   serverSocket.receive(receivePacket) ;
```

```
    String sentence = new String(receivePacket.getData()) ;

    if (sentence.substring(0,9).equals("NSAddress")) {
     InetAddress IPAddress = receivePacket.getAddress() ;
     int port = receivePacket.getPort() ;
     sendData = NSAddress.getBytes() ;
     System.out.println(sendData.length) ;
     DatagramPacket sendPacket = new DatagramPacket
     (sendData, sendData.length, IPAddress, port) ;
     serverSocket.send(sendPacket) ;
    }
   }
  }
}
```

Le lancement du résolveur peut s'effectuer de la manière suivante :

```
Resolver resolver = new Resolver("corbaloc : :192.168.1.2 :2001
                                                    /NameService");
resolver.runServerUDP() ;
```

L'adresse *corbaloc : :localhost :2001/NameService* spécifie que le naming service s'exécute sur la machine désigné par l'adresse IP *192.168.1.2* et sur le numéro de port *2001*.

L'explorateur de naming service

Nommée *NSTool*, cette classe est utilisée par les composants et les connecteurs afin de lister les noms d'objets distribués dont leur référence à été enregistrée dans un naming service. NSTool fournit aussi un service permettant de retourner la référence d'un *naming context*.

La premier service fourni explore un *naming context* donné :

```
public Vector listObjects(String path,NamingContextExt NC, String
                                                          name) {
 Vector result = new Vector() ;
 NamingContextExt naming = getNamingContext(path,NC) ;
 BindingListHolder bl = new BindingListHolder() ;
 BindingIteratorHolder blIt= new BindingIteratorHolder() ;
 naming.list(10, bl, blIt) ;
 Binding bindings[] = bl.value ;
 for (int i=0 ; i < bindings.length ; i++) {
  int lastIx = bindings[i].binding_name.length-1 ;
  if ((bindings[i].binding_name[lastIx].id).compareTo(name) == 0){
   result.add(bindings[i].binding_name[lastIx].kind) ;
  }
 }
 return result ;
}
```

Le second service renvoie la référence du *naming context* correspondant à un chemin d'accès donné où chaque module est séparé par le caractère "/" :

```
public NamingContextExt getNamingContext(String path,
                                NamingContextExt NC) {
 NameComponent[] componentName=null ;
 NamingContextExt naming=null ;
 String name="" ;
 for(int j=0 ;j<path.length() ;j++) {
  if(path.charAt(j) ==  /  ) {
   componentName = NC.to_name(name+".cont");
```

```
    naming = NamingContextExtHelper.narrow
                (NC.resolve(componentName));
    NC = naming ;
    name="" ;
   }
   else {
    name = name + path.charAt(j);
   }
  }
  return NC ;
}
```

L'initialisation du naming service

Cette classe est générée à partir du référentiel. Le prototype produit, pour chaque noeud rencontré, un *naming context*. La difficulté de cette tâche réside dans la génération des liens hiérarchiques entre les différents *naming contexts*. La technique utilisée dans notre prototype est simple : une variable de type *NamingContext* est déclarée à chaque nouveau noeud rencontré et porte le nom du type de composant qui lui est associé. Les liens entre les contextes peuvent être générés sur base de la connaissance du nom du noeud parent. La méthode suivante est appelée de manière récursive à chaque noeud :

```
void createContext(ComponentType node, FileCreator file) {
 file.write(" org.omg.CosNaming.NamingContext naming"+node
                                          .getName() +" ;") ;
 file.write(" org.omg.CosNaming.NameComponent[] nameComponent"+node
 .getName()+ "= new org.omg.CosNaming.NameComponent[1] ;") ;
 file.write(" nameComponent" + node.getName() + "[0] = new org.omg
 .CosNaming.NameComponent() ;") ;
 file.write("nameComponent" + node.getName() + "[0].id = \"module"
                                        + node.getName() + "\" ;");
 file.write("nameComponent" + node.getName() + "[0].kind
                                            = \"cont;");
 if(node.getParent() == null)
 {
  file.write("naming"+node.getName()+"=rootCtx.new_context();");
  file.write(" try {") ;
  file.write(" rootCtx.bind_context(nameComponent" + node.getName()
 + ",naming" +node.getName() +") ;") ;
  file.write(" } catch(Exception e) { }") ;
 }
 else
 {
  file.write(" naming" + node.getName()+"=naming"+node.getParent()
  .getName() + ".new_context() ;") ;
  file.write(" try {") ;
  file.write(" naming" + node.getParent().getName() + ".bind_context
  (nameComponent" +node.getName()+ ",naming"+node.getName() +");");
  file.write(" } catch(Exception e) { }") ;
 }
}
```

Cependant cette solution comporte une faille lorsque plusieurs types de composants possèdent un même nom dans des contextes tout à faits différents. En effet, notre prototype, dans ce cas, générerait plusieurs attributs *NamingContext* de même nom, ce qui est interdit en Java. Une autre solution est d'ajouter un attribut de type *NamingContext* dans la classe *ComponentType* du métamodèle. Il suffirait dès lors de parcourir le référentiel et de lier les attributs *NamingContext* entre eux selon les liens hiérarchiques existant entre les différents noeuds.

Le déploiement d'un composant

Les classes de déploiement des composants ont pour rôle de permettre le lancement, à tout moment du cycle de vie d'un système, d'un composant d'un certain type. Le code généré obtient la référence d'une factory capable d'instancier un composant d'un certain type et exécute les méthodes d'initialisation. Une classe spécifique, nommée *runSystem.java*, est générée par le prototype et est utilisée pour l'amorçage du système.

L'exemple ci dessous représente une partie de code contenue dans une classe de déploiement générée par notre prototype permettant de déployer un type de composant *Application*.

```
try {
 NameComponent[] nameFactory = new NameComponent[1] ;
 nameFactory[0] = new NameComponent("factoryApplication",
 (String)gui.getCombo().getSelectedItem() ) ;
 NamingContextExt naming = ns.getNamingContext("moduleApplication/"
 ,rootCtx) ;
 FactoryApplication factory ;
 factory = FactoryApplicationHelper.narrow(naming.resolve(name
 Factory)) ;
 Application instance = factory.createApplication() ;
 instance.initSubComponents() ;
 instance.initConnectors() ;
}
catch(Exception ex) {
 ex.printStackTrace() ;
}
```

où la séquence d'instruction *(String)gui.getCombo().getSelectedItem()* retourne une adresse IP choisie par l'utilisateur depuis une interface graphique.

4.5 Déploiement

Nous venons de détailler les différentes étapes de fonctionnement de notre prototype. Dans cette section, nous allons montrer comment déployer un système à partir des classes générées par le prototype.

Voici dans l'ordre, la succession des tâches à exécuter pour déployer l'application *client-serveur* dont l'architecture a été définie en MADL :

1. **Lancement du naming service** sur un ordinateur du réseau en spécifiant un numero de port. Nous avons choisi le port *2001*.
2. **Exécution du résolveur** sur un ordinateur quelconque du réseau.
3. **Initialisation du naming service**. Cette classe peut être exécutée à partir de n'importe quelle machine.
4. **Lancement des serveurs** hébergeant les différentes factories. Il s'agit des classes *StartFactoryApplication.java*, *StartFactoryClient.java* et *StartFactoryServer.java*. Ces classes sont exécutées selon les informations contenues dans une matrice de déploiement.
5. **Amorçage du système** en exécutant la classe *RunSystem.java*.
6. **Déploiement d'un composant de type client** en exécutant la classe *DeployClient.java*

Le système s'initialise par récursivité comme nous l'avons expliqué dans le chapitre précédent. Lorsque nous déployons un composant de type *Client*, une interface graphique propose à l'utilisateur de choisir la machine sur laquelle ce nouveau composant doit être créé. Cette interface graphique ressemble à ceci :

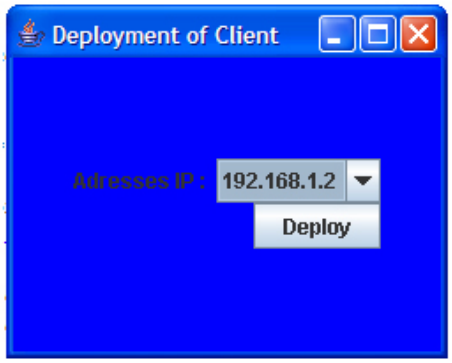

FIG. 4.2 – Déploiement d'un composant de type *Client*.

Pour tester le système il suffit de fournir une implémentation pour le service *reverse()* défini dans l'interface *Business* et d'ajouter l'instruction suivante dans le classe de déploiement du client :

```
instance.link().reverse("une chaîne à renverser");
```

Nous aurions pu générer des interfaces graphiques d'utilisation des services fournis par un composant. Cependant, cela n'est pas l'objectif de ce mémoire.

4.6 Conclusion

Dans le chapitre précédent, nous avons élaboré des pistes de solutions permettant de doter une spécification d'architecture de certains mécanismes. Ces mécanismes définissent le comportement des composants d'un système et fournit des moyens qui autorisent leur interconnexion. Le développement de ce prototype montre qu'il est possible, à partir du métamodèle défini, d'implémenter ces mécanismes dans un langage de programmation particulier et valide donc la faisabilité de leur mise en place.

Chapitre 5

Discussions : Reconfiguration dynamique d'une architecture

5.1 Problématique

De plus en plus d'applications réparties nécessitent une exécution continue 24 heures sur 24. Lorsque ces applications doivent être arrêtées, pour des raisons de maintenance par exemple, des coûts très importants peuvent être engendrés. Ces coûts sont liés à l'arrêt de l'application, à l'installation de nouvelles versions ou encore aux difficultés de les faire évoluer.

L'enjeu de la reconfiguration dynamique est de répondre à cette problématique en fournissant des mécanismes permettant de faire évoluer ou de modifier une application au cours de son exécution.

[12] définit la reconfiguration dynamique d'applications réparties comme un ensemble de changements apportés à une application répartie pendant son exécution. Les modifications peuvent être de quatre sortes :

* Modifications **structurelles** ayant pour but de modifier la structure d'une application. Il s'agit par exemple d'ajouter un composant ou de supprimer une connexion.

* Modifications **géographiques** d'une application visant à changer la distribution des composants.

* Modifications de **mise en oeuvre** d'un composant. Ces modifications peuvent par exemple remplacer le code d'implémentation d'un composant à son exécution.

* Modifications **des services** fournis par un composant. Ce type de changement permet d'ajouter un nouveau service ou de modifier la signature d'un service fourni par un certain composant par exemple.

L'objectif de ce chapitre est de proposer des mécanismes de reconfiguration de la structure d'une architecture définie en MADL. Nous étudierons donc uniquement le premier type de changement. Nous allons tout d'abord déterminer les problèmes généraux liés à la reconfiguration dynamique de la structure d'une application. Ensuite, nous présentons brièvement les approches existantes en matière de reconfiguration structurelle. Nous proposons une solution de reconfiguration pour le langage MADL. Finalement nous critiquons nos règles de mapping dans le cadre de la mise en oeuvre de certaines reconfigurations.

5.2 Les problèmes généraux

Fournir des mécanismes de reconfiguration nécessite la prise de conscience des problèmes engendrés par une reconfiguration. C'est pourquoi [13] distingue quatre problèmes généraux liés à la reconfiguration dynamique d'une architecture. Nous passons ici en revue ces quatre problèmes.

5.2.1 Modification des références

Pour qu'un composant puisse communiquer avec un autre, il faut qu'il possède sa référence. Or, lors d'une reconfiguration d'une connexion, cette référence est susceptible d'être modifiée pour permettre à ce composant de communiquer avec un nouveau composant. Les mécanismes de reconfiguration doivent être capables de mettre à jour ces références pendant l'exécution d'une application.

Dans le cadre du langage MADL, cela signifie que les mécanismes de reconfiguration doivent être capables de mettre à jour les références des connecteurs out détenus par les connecteurs in.

5.2.2 Gestion des messages en transit

Lorsque certains composants sont supprimés, certains messages qui leur étaient destinés avant leur suppression peuvent rester en transit dans les canaux de communication. La reconfiguration dynamique doit pouvoir récupérer ces messages et assurer que leur ordre d'arrivée ne soit pas modifié.

5.2.3 Préservation de l'état d'un composant

Lorsqu'un composant est supprimé, il se trouve dans certain état. Un état désigne l'ensemble des informations internes à un composant permettant son bon fonctionnement. Les mécanismes de reconfiguration doivent être capables de restituer l'état d'un composant supprimé vers de nouveaux composants ajoutés afin qu'ils puissent commencer leur exécution dans un état cohérent par rapport au reste de l'application.

5.2.4 Validation des changements

Lors de l'exécution d'un mécanisme de reconfiguration, il faut s'assurer que l'application reste cohérente. En ce qui concerne MADL, cela signifie que les contraintes liées aux structures du langage doivent être vérifiées et validées avant de pouvoir appliquer une reconfiguration. Par exemple, l'ajout d'un nouveau composant ne peut s'exécuter que si le nombre de composants déjà présents est inférieur à la cardinalité maximum de l'ensemble d'instances.

5.3 Les approches existantes

Dans [13], une étude bibliographique est menée et met en évidence trois recherches intéressantes proposant des solutions pour la reconfiguration dynamique d'applications réparties. Il s'agit de *CONIC*, *POLYLITH* et *ARGUS*. Nous présentons dans cette section ces trois approches de base.

5.3.1 CONIC

CONIC permet la construction d'applications réparties sur base d'un modèle de composants et d'un langage de configuration visant à décrire une application comme un assemblage de composants.

CONIC utilise un configurateur qui, à partir d'ordres de reconfiguration, de la description structurelle de l'application et des structures d'exécution, est capable de modifier les interconnexions courantes et de modifier l'état des composants. Le configurateur permet aussi d'installer les différents composants sur des sites et de créer les connexions entre eux. Il utilise pour cela des primitives, qui peuvent aussi

être utilisées à des fins de reconfiguration.

CONIC permet uniquement le changement de la structure d'une application à partir des quatre primitives suivantes :

* **link** : crée une connexion entre deux composants.
* **unlink** : détruit une connexion entre deux composants. *CONIC* s'assure que les canaux de communication sont vides avant de déconnecter deux composants.
* **add** : ajoute un nouveau composant. Ce dernier est considéré comme isolé lors de son ajout et n'implique donc aucune vérification à la reconfiguration.
* **remove** : détruit un composant existant. Cette manipulation impose au configurateur de vérifier qu'aucun message, concernant ce composant, n'est en transit sur le réseau de communication.

L'algorithme de reconfiguration défini par *CONIC* repose sur la définition d'états abstraits de composants. Un composant peut posséder trois états abstraits :

* **actif** : le composant fonctionne normalement, peut générer des requêtes et en traiter.
* **passif** : le composant ne peut plus exécuter de requêtes et aucune requête dont il est l'initiateur n'est en cours. Il peut cependant en recevoir. Dans cet état, le canal de communication de sortie est vide.
* **gelé** : dans cet état, le composant est bloqué. Tous les canaux de communication liés à ce composant sont vides. Cet état nécessite que tous les composants qui lui sont connectés soient dans un état passif.

Le configurateur peut forcer un composant à passer à un de ces états via les primitives *activate*, *passivate* et *freeze*. L'algorithme de validation d'un ordre de configuration revient donc à déterminer l'ensemble des composants à faire passer dans l'état passif ou gelé afin de pouvoir effectuer cette reconfiguration. Par exemple, un ordre de suppression d'une connexion impose à tous les composants qui en font partie d'être gelés.

Cette approche peut représenter un inconvénient lorsque trop de composants doivent passer à l'état passif afin d'exécuter un ordre de reconfiguration. C'est le cas lorsqu'il existe trop de dépendances entre les composants.

5.3.2 POLYLITH

POLYLITH se base sur un MIL (Langage d'Interconnexions de Modules) afin de générer des applications réparties. Son approche est différente de celle adoptée par *CONIC* car elle définit dans chaque composant des informations permettant sa reconfiguration. De plus, les mécanismes de reconfiguration sont implémentés, non plus dans un configurateur, mais dans le bus de communication, ce qui facilite l'interception des messages en transit. Les types de changement supportés sont la modification de structure et d'implémentation.

La préservation de l'état d'un composant est en fait à la charge du programmeur. Un composant reconfigurable doit implémenter une fonction d'encodage et de décodage. La fonction d'encodage vise à sauvegarder, sous un format choisi par le programmeur, l'état du composant. La fonction de décodage permet à un autre composant de décoder un état sauvegardé afin de préserver l'état d'un composant supprimé par exemple.

L'algorithme de reconfiguration implémenté par *POLYLITH* bloque les canaux de communication afin de permettre l'exécution des ordres de reconfiguration. La notion d'état abstrait n'est pas nécessaire vu que la gestion de la reconfiguration s'opère au niveau du bus de communication qui est capable

d'intercepter tous les messages en transit avant de bloquer un canal de communication. La gestion des références est aussi à la charge du bus de communication. L'avantage réside dans le fait qu'uniquement les canaux de communication concernés sont bloqués.

5.3.3 ARGUS

ARGUS est outil de génération d'applications réparties gérant des informations persistantes avec accès concurrents. Son approche se différentie des deux précédentes dans le sens que les mécanismes de reconfiguration sont fournis par un système d'exploitation fonctionnant sur base de propriétés transactionnelles.

ARGUS ne nécessite pas le blocage des composants ou des canaux de communication car il utilise le système de transaction pour garder le système dans un état cohérent. Le retour en arrière est utilisé pour reconfigurer une application en se basant sur le concept d'**action**. Une action est une transaction, une partie de code qui s'exécute complètement ou pas du tout (*propriété d'atomicité*). Les actions s'exécutent de manière sérialisée. *ARGUS* permet d'ajouter, de supprimer ou remplacer un composant.

Reconfigurer une application avec *ARGUS* revient à annuler les transactions non encore validées. Les composants sont ensuite modifiés à partir d'un état qui est cohérent (garanti par la propriété d'atomicité) en fonction de l'exécution des primitives de reconfiguration. Cet aspect de la reconfiguration peut être utilisé pour la gestion de la tolérance aux pannes. Cependant, ce système impose un modèle de programmation spécifique qui n'est pas utilisable par toutes les applications.

5.4 Solutions de reconfiguration pour le langage MADL

Dans cette section, nous nous inspirons de notre recherche bibliographique pour élaborer des mécanismes de reconfiguration dynamique dans le cadre de la définition du langage MADL. Malgré que *ARGUS* possède de nombreux avantages, nous nous inspirerons uniquement des deux premières méthodes vu que notre approche conceptuelle de déploiement d'une application MADL ne fournit aucune propriété transactionnelle.

Notre solution s'inspire donc en partie de *CONIC* et *POLYLITH* en s'adaptant aux règles de mapping dégagées lors du chapitre 3. Nous apportons ensuite une critique de ces règles de mapping dans le cadre de la mise en oeuvre de mécanismes de reconfiguration dynamique. Nous essayerons finalement d'apporter des solutions aux limitations liées à cette critique.

5.4.1 Principe de reconfiguration dynamique

Notre approche emprunte à *CONIC* l'idée du configurateur et à *POLYLITH* le concept d'encodage et de décodage de l'état d'un composant. Cependant nous adaptons la notion de configurateur au mécanisme de composition d'architecture pour mettre en évidence l'originalité de notre approche.

Chaque composant composite est en fait un configurateur responsable de configurer l'ensemble des composants et des connexions qu'il contient. Comme dans *CONIC*, un composant composite possède une représentation des structures d'exécution qui sont propres à son contexte. Par exemple, le composant *Application* possède une représentation des connexions qui existent entre un *Client* et un *Server*. Cette représentation est utilisée par le configurateur pour permettre l'exécution des primitives de reconfiguration. Cette représentation contient aussi les références vers les sous-composants. C'est ainsi que la distribution d'un type de composant en objet réparti prend tout son sens.

L'avantage de ce type de configurateur se trouve dans la distribution de la complexité liée à une reconfiguration structurelle. En effet, au lieu d'être à la charge d'une seule entité, les mécanismes de

reconfiguration sont encapsulées dans chaque composant composite du système, ce qui permet aussi une granularité plus fine de la reconfiguration.

Chaque composant composite implémente les quatre primitives de reconfiguration suivantes : **link**, **unlink**, **add** et **remove**. Ces quatre primitives de reconfiguration peuvent aussi être utilisées pour configurer des composants à leur déploiement.

Passons maintenant en revue les quatre problèmes généraux liés aux reconfigurations structurelles et les mécanismes que nous définissons pour exécuter ces reconfigurations. Nous supposons aussi qu'un configurateur sérialise l'exécution des ordres de reconfiguration qu'il reçoit.

5.4.2 Gestion des messages en transit

L'exécution des primitives de reconfiguration ne peut s'effectuer que si le configurateur est capable de bloquer les canaux de communication des composants impliqués dans la reconfiguration. Nous proposons ici deux méthodes différentes inspirées respectivement des approches de *CONIC* et *POLYLITH*.

La première méthode consiste à définir des états abstraits de composants (*actif*, *passif* et *gelé*). Dans ce cas, le configurateur exécute un algorithme de blocage des canaux de communication avant d'effectuer les ordres de reconfiguration. Chaque composant implémente les méthodes activate(), passivate() et freeze() définies dans leur interface IDL. Ces méthodes sont donc accessibles à distance et peuvent être exécutées par le composite.

Dans la seconde méthode, le blocage des canaux de communication sont à la charge des entités qui gèrent la distribution des messages. Cette méthode n'est applicable que lorsque la distribution des messages est implémentée à l'aide d'*event channels* qui garantissent la distribution de tous les messages. Cette solution laisse au configurateur le soin de bloquer les communications à partir des event channels liés aux connexions faisant l'objet d'une reconfiguration. Avant chaque reconfiguration, les canaux, définis dans l'*event channel* et qui sont propres aux composants reconfigurés, sont temporairement bloqués.

5.4.3 Modification des références

L'exécution des primitives *link* et *unlink* nécessitent la mise à jour des références contenues dans les connecteurs in des composants définissant des interfaces clientes. Afin d'exécuter ces ordres de reconfiguration, chaque connecteur in implémente une méthode supplémentaire nommée **reset ()**. Cette méthode a pour but d'effacer les références contenues dans un connecteur in. De plus, la signature de la méthode d'initialisation doit être modifiée pour permettre au composant composite de reconfigurer l'ensemble des références à distance. Ainsi l'opération *init()* prend un argument supplémentaire : un ensemble de chaînes de caractères désignant le nom des composants connectés.

Supposons qu'une connexion, nommée *link*, existe entre un composant *c1* et *s1*. Le configurateur *a* (qui contient *c1* et *s1*) reçoit les ordres de reconfiguration suivants :

* unlink (link,c1,s1).

* link (link,c1,s2).

* link link,c1,s3).

Le configurateur *a* exécute pour cela les actions suivantes :

* Effacement des références dans le connecteur in de c1 en exécutant la méthode *reset()*.

* Réinitialisation des références du connecteur in en exécutant la méthode *init(s2,s3)*.

Les canaux de communication liés aux composants *c1* et *s1* doivent être bloqués pour que ces primitives de reconfiguration puissent être exécutées.

5.4.4 Préservation de l'état d'un composant

La préservation de l'état d'un composant peut être garantie comme dans *POLYLITH*. Ainsi chaque composant implémente deux méthodes supplémentaires : *code()* et *uncode()*. L'implémentation du codage et décodage de l'état d'un composant est bien sûr laissée au programmeur. Dans notre approche, le composant composite code l'état d'un composant avant de le supprimer. Il transfère cet état vers un nouveau composant sur lequel la méthode de décodage est exécutée.

5.4.5 Validation des changements

Le composant composite doit être capable de vérifier certaines contraintes avant de valider un ordre de reconfiguration. La représentation des structures d'exécution définie dans chaque configurateur est complétée des contraintes leur étant associées. C'est ainsi qu'un composant composite connaît :

* les **cardinalités** minimum et maximum des ensembles d'instances faisant partie de cette représentation.
* les **cardinalités** minimum et maximum des contrats faisant partie de cette représentation.

Lorsqu'un configurateur reçoit un ordre de configuration, il vérifie si aucune des contraintes n'est susceptible d'être violée par l'exécution de cet ordre, auquel cas ce dernier se verrait annulé.

5.4.6 Interface de reconfiguration

Si la distribution des mécanismes de reconfiguration est bien effectuée sur l'ensemble des composites d'une application, un moyen central d'accès à tous ces points de reconfiguration doit cependant être défini.

Nous allons pour cela utiliser les dépendances hiérarchiques qui existent entre les composants d'un système. En effet, chaque composant est capable de renvoyer la référence de ses sous-composants. A partir de la référence racine d'une application, il est donc possible d'accéder à l'ensemble des points de reconfigurations du système.

Le composant d'amorçage d'un système (la classe *RunSystem* dans le chapitre 4) est la seule entité à connaître la référence du composant racine étant donné que c'est ce composant qui démarre le déploiement et la création de l'ensemble des composants. Configurer une application nécessite donc l'accès à la référence de ce composant d'amorçage. La solution est d'enregistrer ce composant dans le *naming service* sous un nom respectant une convention spécifique. Le programme suivant reconfigure le contenu du composant racine *Application* :

```
BootstrapComponent b = nameService.resolve("BootstrapComponent") ;
Application application = b.getRootComponent() ;
application.addComponent("Server","s2") ;
application.unlink("link","c1","s1") ;
application.link("link","c1","s2") ;
```

5.5 Les limites des règles de mapping

Nous venons de définir des mécanismes de reconfiguration capables de modifier les structures d'exécution d'un système à son exécution, c'est à dire les instances et les connexions qui peuvent exister entre elles. Cependant, notre configurateur n'est pas capable de connecter une instance avec un nouveau type de composant. Prenons l'exemple d'une description d'architecture qui spécifie une instance *a* de type *A* connectée à une instance *b* de type B grâce au connecteur *link*. Il serait impossible au configurateur d'exécuter l'ordre de reconfiguration suivant : link("link","a","c") où *c* représente une instance de type *C*. Une autre limite concerne l'ajout de nouveaux types de composants au sein d'un composite.

Cette problématique provient des limites de notre démarche élaborée au chapitre 3. Pour invoquer une opération sur un objet, un composant doit faire appel, et être lié statiquement, au stub correspondant. Dans notre cas, nous générons, à partir des interfaces IDL, tous les *stubs* permettant aux composants d'utiliser les connexions potentielles définies dans la description d'architecture. Cependant, un composant ne peut pas communiquer avec d'autres types de composants car ils ne sont pas liés aux stubs correspondants. Si nous reprenons notre exemple ci dessus, nous constatons que le configurateur ne peut exécuter cet ordre car *a* ne possède pas de *stubs* afin d'exécuter des appels de méthodes sur *c*.

Dans la suite de cette section, nous utilisons Fractal RMI comme point de référence pour apporter une solution à cette problématique.

5.5.1 Fractal RMI

Fractal RMI[14] est un ensemble de composants Fractal qui fournissent des factories afin de créer des liens distribués entre des composants Fractal. Fractal RMI fournit un composant de gestion de protocoles, un composant de liaison et une factory de génération de stubs pour exécuter les appels à distance entre les composants Fractal.

Chaque interface d'un composant Fractal est un point accessible à distance. Le composant d'amorçage est lui-même accessible à distance. Ce composant fournit une factory pouvant être invoquée à distance pour déployer et instancier des composants sur différents sites. Fractal RMI possède un langage ADL qui permet de représenter la description d'une architecture sous forme d'arbre et de noeuds virtuels. Ces noeuds sont ensuite parsés pour déployer, instancier et lier les composants entre eux.

Nous pouvons constater qu'il existe une certaine similitude entre l'approche adoptée par Fractal RMI et les solutions de déploiement que nous avons mises en oeuvre dans les chapitres précédents. Il est dès lors intéressant d'observer de plus près les mécanismes utilisés par Fractal RMI pour distribuer les liens entre composants.

Comme nous pouvons le voir sur la figure suivante, Fractal RMI utilise des factories de *stubs* et des factories de *skeletons* afin de créer dynamiquement les classes correspondant aux *stubs* et *skeletons* d'une nouvelle interface exportée durant l'exécution. Ces factories se basent sur ASM[15] qui est un environnement de manipulation de code Java utilisé pour générer des classes dynamiquement ou pour modifier des classes en cours d'exécution. Ainsi les *stubs* et *skeletons* ne doivent pas être générés de manière statique.

5.5.2 Solution : Création dynamique de *stubs*

En nous basant sur le modèle adopté par Fractal RMI, nous pouvons résoudre notre problème en fournissant des mécanismes de génération dynamique de *stubs*. Cela peut s'effectuer par l'intermédiaire d'une factory de *stubs* qui est capable de générer un *stub* responsable d'un type d'objet réparti avant de le charger en mémoire. Ensuite le configurateur doit être capable de changer l'implémentation d'un

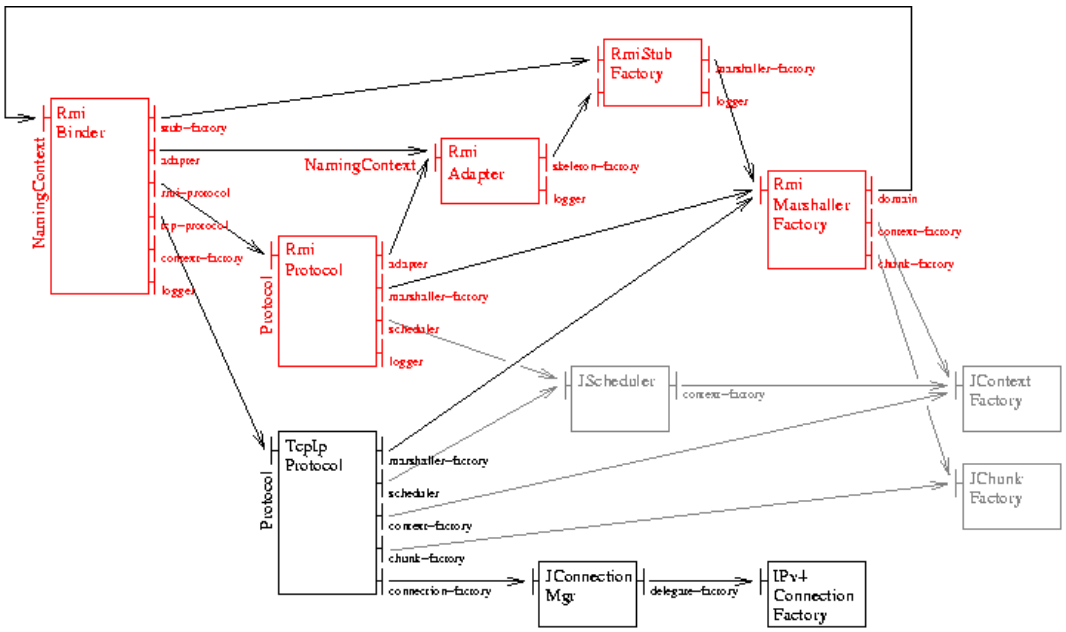

FIG. 5.1 – Architecture du modèle Fractal RMI.

composant pendant son exécution afin qu'il puisse utiliser les *stubs* qui viennent d'être générés.

Si nous reprenons notre petit exemple exposé au début de cette section, le mécanisme de reconfiguration serait le suivant :

* Le configurateur génère, via la factory, un *stub* capable d'envoyer des requêtes vers *c*.

* Le configurateur modifie la classe représentant l'objet *a* et ajoute un attribut représentant le *stub*.

* Le configurateur exécute l'ordre de reconfiguration comme nous l'avons expliqué.

Cette solution est uniquement conceptuelle et pourrait faire l'objet d'une étude entière spécifique. C'est pourquoi nous n'irons pas plus loin dans les détails.

Conclusion

Ce mémoire représente bien une contribution à la définition et au développement du langage MADL. En effet, nous avons, d'une part, associé une des sémantiques possibles aux structures du langage et, d'autre part, validé les choix conceptuels opérés à travers une étude de cas. Cette étude de cas, consistant à élaborer des pistes de Mapping vers le middleware Corba, a non seulement validé les différentes sémantiques attribuées aux concepts du langage mais a aussi permis de fournir à MADL des mécanismes de déploiement d'une architecture sur un système distribué.

Ces mécanismes reposent essentiellement sur la composition hiérarchique et la localisation de composants à travers des services de nommage. En effet, notre approche dote les composants composites d'un rôle important dans le déploiement et la réconfiguration d'une architecture. En plus de représenter un moyen de définition de différents niveaux d'abstraction, ils déploient les sous-composants à l'initialisation de l'application et permettent de reconfigurer ces sous-composants à l'exécution. L'utilisation de *naming services* et factories jouent un précieux rôle dans l'instanciation des composants à distance et dans la localisation des connecteurs permettant de mettre les composants en communication.

En dévelopant un prototype d'implémentation en Java, nous avons prouvé que nos différentes solutions étaient tout à fait réalisables et nous avons validé notre contribution au développement de MADL.

Le dernier chapitre apporte sa contribution sur le plan de la reconfiguration dynamique, qui est une caractéristique de plus en plus étudiée par les concepteurs de systèmes distribués afin de limiter les couts liés à leur maintenance. Certains aspects de la reconfiguration dynamique nous ont permis de constater que notre approche comportait des limites. Ces limites peuvent être contournés par génération dynamique de *stubs* au sein des composants.

Cependant, ce mémoire représente uniquement une première approche consistant plutôt à résoudre les problèmes généraux rencontrés lorsque l'on définit un langage de description d'architecture. C'est ainsi que ce travail pourrait être utilisé comme *input* à d'autres travaux plus spécifiques. Il pourrait par exemple s'agir de :

* Fournir une **spécification complète** du language en termes de sémantique et d'implémentation vers un langage de programmation particulier. Un tel travail pourrait servir de référénce pour le développement d'un compilateur.

* Elaborer des **solutions permettant la génération dynamique de stubs** et le remplacement de code à l'éxecution d'un composant. En effet, au cours du dernier chapitre, nous avons uniquement présenté une solution conceptuelle. Celle-ci pourrait être validée et completée par une étude supplémentaire. Ce mémoire pourrait aussi être utilisé pour implémenter nos solutions de reconfiguration dynamique afin d'en évaluer les performances.

Bibliographie

[1] Mary Shaw, David Garlan Software Architecture : Perspectives on a Emerging Discipline, 1996

[2] Len Bass, Paul Clements, Rick Kazman, Software Architecture in Practice, 2003

[3] N.Medvidovic, R. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, Janvier 2000

[4] Projet ACCORD, Etat de l'art sur les Langages de Description d'Architecture (ADLs), 2002

[5] Magee J, Dulay N, Kramer J, Regis : A Constructive Development Environment for Distributed Programs, 1994

[6] E.Bruneton, T.Coupaye, J.B. Stefani, The Fractal Model Specification, Online documentation http ://fractal.objectweb.org/specification/, Februari 2004

[7] T. Coupaye, V.Quéma, L. Seinturier, J.-B. Stefani, Intergiciel et Construction d'Applications Réparties, 2006

[8] V. Englebert, F. Vermaut, Attribute-Based Refinement of Software Architectures

[9] G. Booch, C. Jacobson, and J. Rumbaugh, The Unified Modeling Language - a reference manual. Addison Wesley, Addison-Wesley, 1998

[10] Object Management Group, The Common Object Request Broker : Architecture and Specification, 2.4 ed., Oct. 2000

[11] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John, Design Patterns : Elements of Reusable Object-Oriented Software, 1994

[12] J. Kramer, J. Magee, The Evolving Philosophers Problem : Dynamic Change Management, Novembre 1990

[13] Noël De Palma, Reconfiguration Dynamique D'applications Réparties, 1998

[14] ObjectWeb, Fractal RMI Documentation, Online documentation http ://fractal.objectweb.org/current/doc/javadoc/fractal-rmi/overview-summary.html

[15] Eric Bruneton, Romain Lenglet, Thierry Coupa ASM : a code manipulation tool to implement adaptable systems